ЭВМ

В ПРОИЗВОДСТВЕ

С.Н. БАРАНОВ
Н.Р. НОЗДРУНОВ

ЯЗЫК ФОРТ И ЕГО РЕАЛИЗАЦИИ

МАШИНОСТРОЕНИЕ



С.Н.БАРАНОВ
Н.Р. НОЗДРУНОВ

ЯЗЫК ФОРТ И ЕГО РЕАЛИЗАЦИИ



Ленинград
«Машиностроение»
Ленинградское отделение
1988

ББК 32.973
Б24
УДК 681.3.06

Рецензент канд. техн. наук Л. Г. Осовецкий

Баранов С. Н., Ноздрунов Н. Р.

Б24 Язык Форт и его реализации.— Л.: Машиностроение. Ленингр. отд-ние, 1988.— 157 с, ил. (ЭВМ в производстве.)

ISBN 5-217-00324-3

Книга является первой крупной отечественной публикацией по языку Форт. Этот язык, получивший широкое распространение за рубежом (особенно как средство программирования для персональных ЭВМ), стал привлекать внимание и советских программистов благодаря особенностям своей методологии. Язык Форт сочетает в себе достоинства интерпретирующих и компилирующих систем и ориентирован на диалоговый режим работы. В книге приведено большое количество примеров.

Книга рассчитана на широкий круг инженеров-программистов и может быть полезна пользователям электронно-вычислительной техники, не имеющим специальной программистской подготовки.

Б $\frac{2405000000\text{-}942}{038\ (01)\text{-}88}$ **КБ-42-10-87** **ББК 32.973**

ISBN 5-217-00324-3

ПРЕДИСЛОВИЕ

Язык программирования Форт (англ. *forth* — вперед и одновременно сокращение от *fourth* — четвертый), которому посвящена эта книга, появился в начале 1970-х гг. в США. Его изобретатель Чарльз Мур первоначально применил его для разработки программного обеспечения микроЭВМ, управляющей работой радиотелескопа [27]. Преимущества работы с языком Форт вместо применявшегося ранее Ассемблера были настолько велики, что вскоре он стал использоваться и на других специализированных ЭВМ.

Быстрый рост популярности языка Форт начался с середины 1970-х гг., когда появились персональные ЭВМ. Оказалось, что этот язык позволяет обходиться сравнительно небольшим набором возможностей первых персональных ЭВМ, превращая их в удобный и эффективный инструмент для самой разной работы. К середине 1980-х гг. Форт выдвинулся на третье место после языков Бейсик и Паскаль в качестве средства программирования для персональных ЭВМ, и рост его применения продолжается [7, с. 54]. Широкое распространение получили коммерческие программные продукты, написанные на Форте: системы обработки текстов, пакеты машинной графики, трансляторы, видеоигры [24, 30]. Стихийно быстрое распространение Форта и его практический успех обусловили необходимость стандартизации языка. В 1983 г. был опубликован стандарт "Форт-83" [3, 23, 26], в соответствии с которым ведется изложение материала в этой книге.

Едва появившись, Форт вызвал ожесточенные споры среди профессионалов-программистов, обсуждавших, в частности, является ли Форт еще одним языком (если языком, то какого уровня — высокого или низкого), операционной системой, интерпретатором или компилятором. Одни считали Форт шагом вперед в раз-

витии программирования, другие — ошибочным выбросом в сторону. К настоящему времени становится ясным, что Форт представляет собой самостоятельный унифицированный подход к разработке программного обеспечения, который действительно позволяет решать практические задачи — от небольших игровых программ до больших систем программного обеспечения, работающих в реальном времени [21; 7, с.56]. Унификация состоит в том, что Форт предполагает последовательное и систематическое использование очень небольшого числа «правил». Например, «мостом» между аппаратурой и прикладной задачей служит всего один язык программирования (Форт), в то время как при традиционном подходе этот разрыв заполняется в несколько приемов разнородными инструментальными средствами (ассемблер, универсальные языки выского уровня, проблемно-ориентированные языки, средства операционной системы).

Система программирования на Форте (форт-система) обычно обеспечивает полный набор средств поддержки для разработки и исполнения программ: операционную систему, интерпретатор для диалогового исполнения, компилятор, ассемблер, текстовый редактор и обслуживающие программы. Все эти компоненты являются расширениями Форта и написаны на том же Форте. Таким образом, Форт является одновременно и системой для пользователя, и собственной метасистемой, т. е. системой, описывающей саму себя (см. приложение 1). В соответствии с общим принципом Форта в форт-системе используется минимум правил и накладывается минимум ограничений, т. е. она обходится почти без синтаксиса, жестко контролируемых интерфейсов для взаимодействия модулей, закрытых для пользователя областей памяти, имеет лишь незначительный по объему встроенный контроль ошибок. Это обеспечивает максимум возможностей для программиста, включая возможность изменять, добавлять или удалять любую часть системы, позволяет расширять систему в заданном направлении и вместе с тем сохраняет ее относительную независимость от аппаратуры.

Разумеется, Форт имеет и свои недостатки. Многие программисты считают, что форт-тексты трудно читаемы из-за применяемой в Форте обратной польской формы и различных неочевидных манипуляций со сте-

ком. В некоторых форт-системах отсутствуют средства для получения независимого программного продукта. Вызывает возражение отсутствие контроля типов при взаимодействии модулей и незащищенность форт-системы от неправильных действий программиста. В то же время становится ясно, что методология Форта находится в общем русле поисков в области технологии программирования, хотя в настоящее время практически отсутствуют работы по методологическим и технологическим аспектам его применения, сравнимые по значимости с исследованиями для традиционных языков [13, 14, 19].

В целом цикл разработки программного продукта остается неизменным: анализ, проектирование, программирование, отладка. Однако лишь на первых двух этапах применяется традиционная технология «сверху — вниз». Программирование и отладка ведутся по методу «снизу — вверх». Благодаря этому отпадает необходимость в модулях-заглушках и в повторных тестированиях всего комплекса программ при заменах заглушек на действительные модули, что сокращает время прохождения всего цикла и позволяет выполнить его несколько раз за то же время. При разработке форт-программ наблюдается тенденция к вычленению относительно замкнутых групп модулей, каждая из которых проходит свой цикл разработки. При этом обычно размер модуля составляет от 1 до 3 строк текста, что резко контрастирует с традиционными языками. Для целей промышленного производства программ сочетание методологии Форта с существующими [14] представляется весьма перспективным, однако практические разработки в этой области пока не известны.

В нашей стране также шли поиски принципов, аналогичных тем, которые ныне определяют язык Форт, в большей степени исходя из теоретических основ программирования [10]. Эти работы привели к созданию интересных систем ДССП [9], КОМФОРТ [12], системы программирования на основе понятия «рабочей смеси» [5, 17] и других. Интерес к языку Форт возрастал по мере получения сведений о нем и достижения собственных результатов в этой области [2,6, 18,20]. Усилиями энтузиастов созданы самостоятельные реализации Форта, которые получают распространение наряду с заимствованными реализациями (см. при-

ложение 2). Язык Форт включается в программное обеспечение школьных компьютеров. Ведутся работы по аппаратной реализации этого языка [12]. В 1985 г. в рамках Рабочей группы по технологии программирования микропроцессорной техники при Комиссии по технологии программирования при ГКНТ была создана целевая подгруппа по языку Форт и родственным системам, задачей которой является обобщение и распространение опыта практического применения этих средств в различных областях.

С 1978 г. в США выходит журнал "Форт Дименшнз" *(FORTH Dimensions)* — основное периодическое издание для массовых пользователей языка Форт. С 1979 г. проводятся ежегодные конференции, материалы которых, отражающие последние достижения в развитии форт-подхода, публикуются в виде сборников. С 1983 г. издается журнал "Джорнал оф Форт Эпликейшн энд Рисёч" *(The Journal of FORTH Application and Research,* шифр ГПНТБ — V1467) — издание для программистов-профессионалов. Журналы «Байт» (BYTE, шифр В1841) и «Д-р Доббз Джорнал» *(Dr. Dobb's Journal,* шифр W9464) посвящают языку Форт специальные выпуски.

До сих пор знакомству широких кругов программистов нашей страны с этим языком препятствовало отсутствие сколько-нибудь обстоятельных публикаций о нем на русском языке. Данная книга является первой такой публикацией и написана с целью дать подробное и по возможности простое введение в язык Форт. Мы надеемся, что знакомство с интересными принципами этого языка позволит читателям по-новому взглянуть на свою программистскую практику и будет полезно во всех отношениях.

Авторы выражают глубокую благодарность Г. С. Кудрявцевой, О. Н. Колесниковой и М. Б. Округину за помощь в подготовке рукописи. Отзывы о книге и предложения можно направлять по адресу: 191065, Ленинград, ул. Дзержинского, 10, ЛО издательства «Машиностроение».

Глава 1

ВВЕДЕНИЕ В ФОРТ

1.1 Основные понятия

Приступая к изучению нового для нас языка программирования, мы прежде всего задаемся вопросами: какие конструкции есть в этом языке (какова морфология), как они записываются (каков синтаксис) и что означают (какова семантика). Например, в широко распространенном языке Паскаль имеется около двадцати конструкций (идентификатор, число без знака, присваивание, условный оператор и др.), синтаксис которых обычно задают с помощью граф-схем или порождающих правил, а семантику объясняют на основе той или иной машиннонезависимой модели вычислений. Часть языка ассемблера любой ЭВМ, предназначенная для записи машинных команд, содержит по одной конструкции на каждую команду. Запись такой конструкции обычно представляет отдельную строку и состоит из мнемонического обозначения команды и размещения операндов, а семантика определяется в терминах реальных действий, которые данная команда выполняет над ячейками памяти, регистрами и другими компонентами архитектуры ЭВМ.

Язык Форт больше всего похож на язык ассемблера. Его синтаксис также максимально прост. Запись каждой конструкции (команды) состоит из одного слова — мнемонического обозначения, в качестве которого может выступать последовательность любых литер, не содержащая пробела. Простота синтаксиса является следствием того, что в качестве вычислительной модели используется стековая машина. Слова-команды этой машины снимают необходимые операнды со стека и оставляют свои результаты (если они есть) также на стеке. Таким образом, программа, написанная на языке Форт, выглядит как последовательность слов, каждое из которых подразумевает выполнение тех или иных действий. Слова разделяются любым числом пробелов и переходов на новую строку; ограничение накла-

дывается только на длину слова — оно должно содержать не более 31 литеры. Стандарт языка определяет сравнительно небольшой набор из 132 «обязательных» слов. Среди них есть слова, позволяющие определять новые через уже имеющиеся и тем самым расширять исходный набор слов-команд в нужном для данной задачи направлении. Некоторые часто требующиеся расширения включены в стандарт в качестве «стандартных расширений» обязательного набора слов.

Вычислительная модель, лежащая в основе языка Форт, состоит из адресного пространства оперативной памяти объемом до 64 К байт, терминала и поля внешней памяти на магнитных дисках объемом до 32 К блоков по 1 К байт каждый. В пределах имеющегося адресного пространства располагаются стек данных и стек возвратов, словарь, буфер для ввода с терминала и буфера для обмена с внешней памятью.

Стек данных обычно располагается в старших адресах оперативной памяти и используется для передачи параметров и результатов между исполняемыми словами. Его элементами являются двухбайтные значения, которые в зависимости от ситуации могут рассматриваться различным образом: как целые числа со знаком в диапазоне от —32768 до +32767, как адреса оперативной памяти в диапазоне от 0 до 65535 (отсюда ограничение 64 К на размер адресного пространства), как коды литер (диапазон зависит от принятой кодировки) для обмена с терминалом, как номера блоков внешней памяти в диапазоне от 0 до 32767 или просто как 16-разрядные двоичные значения. В процессе исполнения слов значения помещаются на стек и снимаются с него. Переполнение и исчерпание стека, как правило, не проверяется; его максимальный объем устанавливается реализацией. Стандарт предусматривает, что стек растет в сторону убывания адресов; это согласуется с аппаратной реализацией стека в большинстве ЭВМ, которые ее имеют.

Стек возвратов по своей структуре аналогичен стеку данных, но используется особым образом некоторыми стандартными словами (подробнее об этом см. в гл. 2).

Начальную часть адресного пространства обычно занимает *словарь* (иначе «кодофайл») — хранилище слов и данных. По мере расширения исходного набора

слов словарь растет в сторону увеличения адресов. Специальные слова из обязательного набора позволяют управлять вершиной словаря — поднимать и опускать ее.

Наряду со стеком данных и стеком возвратов в старших адресах оперативной памяти обычно размещается буфер на 64-100 байт для построчного ввода форт-текста с терминала и буферный пул для обмена с внешней дисковой памятью размером от 1 до 3 и более К байт. Доступ к этим буферам и фактический обмен осуществляют специальные слова из обязательного набора.

1.2. Работа в диалоговом режиме

Программирование на языке Форт является существенно диалоговым. Работая за терминалом, программист вводит слова-команды, а загруженная в память ЭВМ форт-система, т. е. реализация языка Форт на данной ЭВМ, немедленно выполняет обозначаемые этими словами действия. О своей готовности к обработке очередной строки текста форт-система обычно сообщает программисту специальным приглашением (например, знаком < , который печатается на терминале). Получив такое приглашение, программист набирает на терминале очередную порцию форт-текста, заканчивая ее специальным управляющим символом (например, нажимая клавишу "Ввод" или «Перевод строки»). Получив сигнал о завершении ввода, форт-система начинает обработку введенного текста (он размещается в буфере для ввода с терминала), выделяя в нем слова-команды и исполняя их. Успешно обработав весь введенный текст, форт-система вновь приглашает программиста к вводу, и описанный цикл диалога повторяется. Многие форт-системы после успешного завершения обработки выводят на терминал подтверждающее сообщение (обычно ОК — сокращение от английского *o'kay* — все в порядке). Если во время обработки введенного текста выявляется какая-либо ошибка (например, встретилось неизвестное форт-системе слово), то на терминал выводится поясняющее сообщение, обработка введенного текста прекращается и форт-система приглашает программиста к вводу нового текста.

Для завершения работы обычно предусматриваются специальные слова-команды.

Непосредственно вводить с терминала большие форт-тексты неудобно, поэтому их хранят во внешней памяти на дисках, а с терминала программист вводит только слова-команды, отсылающие форт-систему к обработке тех или иных блоков из внешней памяти. Для создания и исправления форт-текстов во внешней памяти используется текстовый редактор, который является специализированным расширением форт-системы.

1.3. Стек данных и вычисления

Как уже говорилось, в основе вычислительной модели для языка Форт лежит стековая машина. Ее команды (слова в языке Форт) обычно используют в качестве своих операндов верхние элементы стека, убирая их со стека и возвращая результаты (если они есть) на место операндов. Как правило, слова используют одно-два верхних значения на стеке. Для их описания будем применять следующую диаграмму:

```
имя    вершина стека до ---> вершина стека после
слова  исполнения  слова       исполнения слова
```

При этом считаем, что самое верхнее значение в стеке (последнее добавленное) находится справа.

Для работы с собственно вершиной стека имеются следующие слова:

```
DUP     A ---> A,A
DROP    A --->
OVER    A,B ---> A,B,A
ROT     A,B,C ---> B,C,A
SWAP    A,B ---> B,A
```

Слово DUP (от DUPLICATE — дублировать) дублирует вершину стека, добавляя в стек еще одно значение, равное тому, которое было до этого верхним. Слово DROP (сбросить) убирает верхнее значение. Слово OVER (через) дублирует значение, лежащее на стеке непосредственно под верхним. Слово ROT (от ROTATE — вращать) циклически переставляет по часовой стрелке три верхних значения в стеке. Наконец, слово SWAP (обменять) меняет местами два верхних значения.

Можно работать с любым элементом стека с помощью слов

```
PICK  An,An-1,...Ao,n  --->  An,An-1,...Ao,An
ROLL  An,An-1,...Ao,n  --->  An-1,...Ao,An
```

Слово PICK (взять) дублирует *n*-й элемент стека (считая от нуля), так что O PICK равносильно DUP, а 1 PICK равносильно OVER. Слово ROLL (повернуть) циклически переставляет *n* верхних элементов стека (тоже считая от нуля) по часовой стрелке, так что 2 ROLL равносильно ROT, 1 ROLL равносильно SWAP, а 0 ROLL является пустой операцией.

Чтобы «увидеть» верхнее значение на стеке, используется слово . (точка) A →, которое снимает значение с вершины стека и печатает его на терминале как целое число в свободном формате (т. е. без ведущих нулей и со знаком минус, если число отрицательно). Вслед за последней цифрой числа слово-точка выводит один пробел, чтобы выводимые подряд числа не сливались в сплошной ряд цифр. Если программист хочет, чтобы напечатанное значение осталось на стеке, он должен исполнить текст DUP .. Слово DUP создаст копию верхнего значения, а точка ее распечатает и уберет со стека.

Перечисленные выше слова работают со значениями, уже находящимися в стеке. А как занести значение в стек? Язык Форт имеет следующее замечательное правило умолчания: если введенное слово форт-системе не известно, то прежде чем сообщать программисту об ошибке, форт-система пытается понять это слово как запись числа. Если слово состоит из одних цифр с возможным начальным знаком минус, то ошибки нет: слово считается известным и его действие состоит в том, что данное число кладется на вершину стека.

Теперь у нас достаточно средств, чтобы привести примеры диалога. Рассмотрим следующий протокол работы:

```
> 5 6 7
 OK
> SWAP . . .
6 7 5  OK
>
```

В ответ на приглашение к вводу (знак > , печатаемый системой) программист вводит три числа: 5, 6 и 7. Обрабатывая введенный текст, форт-система кладет эти числа в указанном порядке на стек и по завершении обработки выводит подтверждающее сообщение ОК и вновь приглашает программиста к вводу. Далее программист вводит текст из четырех слов: SWAP и три точки. Исполняя эти слова-команды, форт-система меняет местами два верхних элемента стека (5, 6, 7→ 5, 7, 6,) и затем поочередно три раза снимает верхнее значение со стека и печатает его. В результате на терминале появляется текст 6 7 5 и сообщение ОК, указывающее на завершение обработки, после чего система вновь выдает программисту приглашение на ввод.

Для внешнего представления чисел используется система счисления, задаваемая программистом. Стандарт языка предусматривает следующие слова для переключения в наиболее общеупотребительные системы:

```
DECIMAL     --->       десятичная
HEX         --->       шестнадцатиричная
OCTAL       --->       восьмеричная
```

Первоначально устанавливается десятичная система. Если в процессе работы будет исполнено, например, слово HEX (от HEXADECIMAL — шестнадцатиричная), то при дальнейшем вводе и выводе чисел будет использоваться шестнадцатиричная система с цифрами от 0 до 9 и от A до F до тех пор, пока основание системы счисления не будет вновь изменено. Внутренним же представлением чисел является обычный двоичный дополнительный код, применяемый в большинстве существующих ЭВМ.

Слова-команды, выполняющие арифметические операции над числами, являются общепринятыми математическими обозначениями:

```
+       A,B  --->  сумма A+B
-       A,B  --->  разность A-B
*       A,B  --->  произведение A*B
/       A,B  --->  частное от A/B
MOD     A,B  --->  остаток от A/B
/MOD    A,B  --->  остаток от A/B, частное от A/B
ABS       A  --->  абсолютная величина A
```

```
NEGATE   A  ---> значение с обратным знаком -A
1+       A  ---> A+1
1-       A  ---> A-1
2+       A  ---> A+2
2-       A  ---> A-2
2/       A  ---> частное от A/2
```

При сложении, вычитании и умножении не учитывается возможность переполнения, в случае его возникновения используются младшие 16 разрядов результата. Такая арифметика называется *арифметикой по модулю 65536* (2 в степени 16); ее основное достоинство состоит в том, что она дает одинаковые в двоичном представлении результаты независимо от того, как понимаются операнды: как числа со знаком в диапазоне от —32768 до +32767 или как числа без знака в диапазоне от О до 65535.

Операции деления / , MOD и /MOD рассматривают свои операнды как числа со знаком. Из нескольких известных математических определений деления с остатком (которые по-разному трактуют случаи, когда операнды имеют разные знаки или оба отрицательны) язык Форт использует так называемое *деление с нижней границей:* остаток имеет знак делителя или равен нулю, а частное округляется до его арифметической нижней границы («пола») [11]. Во многих ЭВМ, имеющих аппаратную реализацию деления, применяются другие правила для определения частного и остатка, однако это обычно не вызывает трудностей при программировании, поскольку самый важный случай с неотрицательными операндами все определения трактуют одинаково.

При выполнении деления возможно возникновение ошибочной ситуации, если делитель — нуль или частное не умещается в 16 разрядов (переполнение, возникающее при делении —32768 на —1).

Одноместные операции ABS и NEGATE игнорируют переполнение, возникающее в том единственном случае, когда операндом является число —32768, возвращая в качестве результата 0.

Наконец, одноместные операции 1+ , 1- ,2+ , 2- выполняют действие, отраженное в их мнемонике: увеличение или уменьшение значения на вершине стека на 1 или 2; аналогично слово 2/ возвращает частное

от деления своего параметра на 2. Эти слова включены в стандарт ввиду частного использования соответствующих действий.

Использование стека для хранения промежуточных значений естественным образом приводит к так называемой *"обратной польской форме"* — одному из способов бесскобочной записи арифметических выражений, подразумевающему постановку знака операции после операндов. Например, выражение (A/B+C) * (D*E — F*(G — H)) записывается следующим образом: A B / C + D E * F G H — * — *. Очевидно, что этот текст выполним для Форта, если A, B и т. д.— слова, которые кладут на стек по одному числу. Таким образом, форт-систему можно использовать как калькулятор. Чтобы вычислить, например, значение (25+18+ + 32) * 5, достаточно ввести такой текст: 25 18 + + 32 + 5 * . . В ответ система напечатает (исполняя точку) ответ —375.

Чтобы повысить точность вычислений в последовательности умножение — деление, стандарт предусматривает два необычных слова:

```
*/     A,B,C  --->  частное от (A*B/C)
*/MOD  A,B,C  --->  остаток, частное от (A*B/C)
```

Особенность этих слов состоит в том, что промежуточный результат A*B вычисляется с двойной точностью и в качестве делимого для C используются все его 32 разряда. Окончательные же результаты являются 16-разрядными числами.

Наряду с описанной выше 16-разрядной арифметикой, язык Форт имеет полный набор средств для работы с 32-разрядными целыми числами через стандартное расширение двойной точности. Внутренним представлением таких чисел является 32-разрядный двоичный дополнительный код, представляющий их как числа со знаком в диапазоне от —2147483648 до + 2147483647 или как числа без знака в диапазоне от 0 до 4294967295. При размещении в стеке число двойной точности занимает два элемента: верхний — старшая половина, предыдущий — младшая. Такое расположение делает простым переход от двойной точности к обычной через слово DROP . Расширение двойных чисел включает следующие слова:

```
2DROP    AA --->
2DUP     AA --->  AA,AA
2OVER    AA,BB --->  AA,BB,AA
2ROT     AA,BB,CC --->  BB,CC,AA
2SWAP    AA,BB --->  BB,AA
D.       AA --->
D+       AA,BB --->  сумма AA+BB
D-       AA,BB --->  разность AA-BB
DABS     AA --->  абсолютная величина AA
DNEGATE AA --->  число с обратным знаком -AA
```

Здесь обозначение типа AA подчеркивает, что значение занимает два элемента стека. Хотя стандартное расширение этого не предусматривает, во многих реализациях языка Форт этот ряд продолжают операции умножения и деления:

```
D*      AA,BB ---> произведение AA*BB
D/      AA,BB ---> частное от AA/BB
DMOD    AA,BB ---> остаток от AA/BB
```

Операндами и результатами перечисленных слов являются значения двойной длины, занимающие по два элемента стека каждый; это обстоятельство отражено в мнемонике, начинающейся с цифры 2, когда два элемента стека выступают как одно целое, или с буквы D (от слова DOUBLE — двойной), когда речь идет об арифметическом значении двойной длины.

Следующие два слова являются переходными между арифметическими операциями одинарной и двойной точности; их мнемоника включает букву M (от слова MIX — смесь) и U (от слова UNSIGNED — беззнаковый):

```
UM*     A,B ---> CC
UM/MOD  AA,B ---> C,D
```

Слово UM* перемножает операнды A и B как 16-разрядные числа без знака, возвращая все 32 разряда получившегося произведения. Слово UM/MOD рассматривает 32-разрядное делимое AA и 16-разрядный делитель B как числа без знака и возвращает получающиеся 16-разрядные остаток C и частное D. Если делитель — нуль или частное превышает 65535, то это рассматривается как ошибка. Для перехода к двойной точности с учетом знака многие реализации имеют слово

$S > D\ A \rightarrow AA$, которое расширяет исходное число A до числа двойной точности распространением знакового разряда.

Чтобы при вводе различать числа двойной и одинарной точности, в сформулированное выше правило умолчания внесена оговорка: если не найденное в словаре слово состоит только из цифр с возможным начальным знаком минус, то это число одинарной точности, а если в слово входят точки (в любом месте), то это число двойной длины. Пример использования форт-системы как калькулятора для работы с числами двойной длины представляет следующий протокол работы:

```
> 1234567.  7654321.  D+  D.
8888888  OK
```

В ответ на приглашение (знак >) программист вводит два числа двойной длины, операцию сложения этих чисел и операцию печати результата. Выполняя указанные действия, форт-система печатает ответ (заметьте, что он уже не содержит точки) и подтверждающее сообщение OK.

1.4. Введение новых слов

Замечательное свойство языка Форт — это возможность вводить в него новые слова, расширяя тем самым набор его команд в нужном программисту направлении. Для введения новых слов чаще всего используется *определение через двоеточие* — определение нового слова через уже известные. Такое определение начинается словом : (двоеточие) и заканчивается словом ; (точка с запятой). Сразу после двоеточия идет определяемое слово, а за ним — последовательность слов, через которые оно определяется. Например, текст : S2 DUP * SWAP DUP * + ; определяет слово S2 , вычисляющее сумму квадратов двух чисел, снимаемых с вершины стека S2 $A,B \rightarrow A{**}2 + B{**}2$. После ввода данного описания слово S2 можно исполнять и включать в описания других слов. При создании таких определений рекомендуется тщательно комментировать все изменения стека. Слово ((открывающая круглая скобка) отмечает начало комментария; все следующие литеры до первой) (закрывающей скобки)

считаются комментарием и при обработке вводимого форт-текста пропускаются.

Перепишем приведенное выше определение слова S2, показывая состояние вершины стека после исполнения каждой строки:

```
S2  ( A,B ---> A**2+B**2 сумма квадратов)
DUP       ( A,B,B)
*  SWAP   ( B**2,A)
DUP *     ( B**2,A**2)
+         ( A**2+B**2)
```

По-видимому, минимальным требованием к документированности определения следует считать задание начального и конечного состояний вершины стека при работе слова.

Рассмотрим подробнее работу форт-системы во время определения новых слов. Мы уже знаем, что получив от программиста очередную порцию входного текста, форт-система выделяет в ней отдельные слова и ищет их в своем словаре. Эту работу выполняет текстовый интерпретатор форт-системы. Если слово в словаре не найдено, то текстовый интерпретатор пытается понять его как число, используя описанное выше правило умолчания. Если слово найдено или оказалось записью числа, то дальнейшие действия интерпретатора зависят от его текущего состояния. В каждый момент времени текстовый интерпретатор находится в одном из двух состояний: в состоянии исполнения или в состоянии компиляции. В *состоянии исполнения* найденное слово исполняется (т. е. выполняется действие, составляющее его семантику), а число кладется на стек. Если же интерпретатор находится в *состоянии компиляции,* то найденное слово не исполняется, а компилируется, т. е. включается в создаваемую последовательность действий для определяемого в данный момент слова. Найденное и скомпилированное таким образом слово будет исполнено наряду с другими такими словами во время исполнения определенного через них слова. Если требуется скомпилировать число, то текстовый интерпретатор компилирует особый литеральный код, который во время исполнения положит значение данного числа на стек.

Проследим за работой текстового интерпретатора по обработке уже рассмотренного определения слова

: S2 DUP * SWAP DUP * + ; . Предположим, что перед началом обработки введенной строки интерпретатор находится в состоянии исполнения. Первым словом является: (двоеточие), которое исполняется. Его семантика состоит в том, что из входной строки выбирается очередное слово и запоминается в качестве определяемого, а интерпретатор переключается в состояние компиляции. Следующие слова, которые интерпретатор будет извлекать из входной строки (DUP , * , SWAP и т. д.), будут компилироваться, а не исполняться, так как интерпретатор находится в состоянии компиляции. В результате с определяемым словом S2 связывается последовательность действий, отвечающая этим словам. Процесс выделения и компиляции слов будет продолжаться до тех пор, пока не встретится ; (точка с запятой). Это слово особенное, оно имеет так называемый «признак немедленного исполнения». Слова с таким признаком исполняются независимо от текущего состояния текстового интерпретатора, поэтому точка с запятой будет вторым исполненным словом после двоеточия. Семантика точки с запятой заключается в том, что построение определения, начатого двоеточием, завершается и интерпретатор вновь переключается в состояние исполнения. Поэтому после ввода определения слова S2 мы тут же можем проверить, как оно работает на конкретных значениях:

```
> 5 4 S2 .
41 OK
```

Слова с признаком немедленного исполнения (другим примером такого слова помимо точки с запятой является открывающая круглая скобка — начало комментария) выполняются по-разному в зависимости от состояния текстового интерпретатора. Некоторые из них даже используются только в одном из этих состояний. Поэтому на диаграмме для таких слов будем отмечать эти случаи, специально указывая состояние компиляции

```
(     --->
      --->   (компиляция)
```

Слово (в обоих состояниях действует одинаково: пропускает все следующие вводимые литеры до закрывающей круглой скобки включительно или до конца

введенной строки, если закрывающая скобка отсутствует.

Слово ; допустимо применять только в состоянии компиляции: ; → (компиляция). Оно завершает построение нового определения и переключает текстовый интерпретатор в состояние исполнения.

Слово IMMEDIATE (немедленный) → устанавливает признак немедленного исполнения для последнего определенного слова. (Подробнее использование этого признака рассматривается в п. 1.7.)

Введенные слова можно исключить из словаря с помощью слова FORGET (забыть) → , которое выбирает из входной строки следующее слово и исключает его из словаря вместе со всеми словами, определенными позже.

Разберем следующий протокол диалога:

```
> 2 2 * .
4 OK
> : 2 3 ;
 OK
> 2 2 * .
9 OK
> FORGET 2
 OK
> 2 2 * .
4 OK
```

Сначала программист вычисляет произведение от умножения 2 на 2 и получает ответ 4. Введя затем определение слова 2 как числа 3, он в дальнейшем получает уже другой ответ. Исключив это определение слова 2 через FORGET, он возвращается к прежней семантике слова 2.

В процессе работы текстового интерпретатора программист может переключать его из состояния компиляции в состояние исполнения и обратно с помощью слов [(открывающая квадратная скобка) → и] (закрывающая квадратная скобка) → . Слово [имеет признак немедленного исполнения и переключает интерпретатор в состояние исполнения, а слово] переключает его в состояние компиляции. Обычно эти слова используются внутри определения через двоеточие, чтобы вызвать исполнение слова или группы слов, не имеющих признака немедленного исполнения. Например, если

в тексте определения понадобилась константа FF00 (в шестнадцатиричной системе), а текущей используемой системой является десятичная, то было бы неправильно включить в текст определения фрагмент HEX FFOO DECIMAL, поскольку слово HEX будет не выполнено, а скомпилировано и число не будет воспринято правильно. Вместо этого следует писать: [HEX] FFOO [DECIMAL] . В языке есть и другие способы, чтобы выразить это же более точно и красиво.

Еще один пример дает использование слова LITERAL (литерал), имеющего признак немедленного исполнения, внутри определения через двоеточие. Оно используется в виде: [(значение)] LITERAL, где (значение) — слова, вычисляющие на вершине стека значение, которое словом LITERAL будет скомпилировано в код как число. Во время исполнения определения это значение будет положено на стек. Таким образом, текст [22*] LITERAL внутри определения через двоеточие эквивалентен употреблению слова-числа 4. Это дает большие возможности для использования констант, «вычисляемых» во время компиляции определения. Аналогичное соответствие имеет место и вне определения через двоеточие, поскольку в состоянии исполнения слово LITERAL не выполняет никаких действий, и поэтому на стеке остается вычисленное перед этим значение.

1.5. Константы и переменные, работа с памятью

Программисту часто бывает удобно работать не с «анонимными» значениями, а с именованными. По аналогии со средствами других языков эти средства языка Форт называются *константами* и *переменными*. Впоследствии мы увидим, что они являются не «изначальными», а (наряду с определениями через двоеточие) частными случаями более общего понятия *«определяющие слова»*.

Слово CONSTANT (константа) A → работает следующим образом. Со стека снимается верхнее значение, а из входного текста выбирается очередное слово и запоминается в словаре как новая команда. Ее действие состоит в следующем: поместить на стек значение A, снятое со стека в момент ее определения. Например, 4 CONSTANT XOP . В дальнейшем при испол-

нении слова XOP число 4 будет положено на стек.

Слово VARIABLE (переменная) A → резервирует в словаре два байта, а из входного потока выбирает очередное слово и вносит его в словарь как новую команду, которая кладет на стек адрес зарезервированной двухбайтной области. Можно сказать, что переменная работает, как константа, значением которой является адрес зарезервированной двухбайтной области.

Работа с переменной помимо получения ее адреса состоит в получении ее текущего значения и присваивании нового. Для этого язык Форт имеет следующие слова:

```
@     A ---> B
!   B,A --->
```

Слово @ (читается «разыменовать») снимает со стека значение и, рассматривая его как адрес области оперативной памяти, кладет на стек двухбайтное значение, находящееся по этому адресу. Обратное действие выполняет слово ! (восклицательный знак, читается «присвоить»), которое снимает со стека два значения и, рассматривая верхнее как адрес области оперативной памяти, засылает по нему второе снятое значение. Эти слова можно использовать не только для переменных, но и для любых адресов оперативной памяти. Следующий протокол работы показывает порядок использования переменной в сочетании с этими словами:

```
> VARIABLE X 1 X !
 OK
> X @ .
1 OK
> X @ NEGATE X ! X @ .
-1 OK
```

В первой строке определяется переменная X, и ей присваивается начальное значение 1. Затем текущее значение переменной X распечатывается. После этого текущее значение меняется на противоположное по знаку и вновь распечатывается.

Полезным вариантом слова ! является слово +! (плюс-присвоить) N,A → , которое увеличивает на N

значение, находящееся в памяти по адресу A. Несмотря на то, что это-слово легко выразить через + и ! :

```
: +! ( N,A ---> ) DUP @ ROT + SWAP ! ;
```

оно включено в обязательный набор слов.

Слова, определенные через CONSTANT и VARIABLE ,— такие же равноправные слова форт-системы, как и определенные через двоеточие. Их также можно использовать в определениях других слов и исключать из словаря словом FORGET.

Для работы со значениями двойной длины имеются слова

```
2CONSTANT    AA --->
2VARIABLE       --->
2@            A ---> BB
2!         BB,A --->
```

из стандартного расширения двойных чисел. При размещении в памяти такие значения рассматриваются как пары смежных двухбайтных значений одинарной длины: сперва (в меньших адресах) располагается старшая половина, затем (в больших адресах) младшая. Адресом значения двойной длины считается адрес его старшей половины.

Константы и переменные позволяют программисту использовать память в словаре вполне определенным образом. А как быть, если требуется что-то иное? Общий принцип языка Форт состоит в том, чтобы не закрывать от программиста даже самые элементарные единицы, из которых строятся его более сложные слова, а предоставлять их наравне с другими словами. Элементарными словами для работы с памятью в словаре, помимо приведенных выше @ и ! , являются следующие:

```
HERE       ---> A
ALLOT   A --->
,       A --->
```

Предполагается, что словарь занимает некоторую начальную часть адресного форт-пространства и у него имеется указатель текущей вершины (словарь растет в сторону увеличения адресов). Слово HERE (здесь) возвращает текущее значение указателя (адрес первого

свободного байта, следующего за последним занятым байтом словаря).

Слово ALLOT (распределить) резервирует в словаре область памяти, размер которой (в байтах) снимает со стека. Резервирование состоит в том, что текущее значение указателя изменяется на заданную величину, поэтому при положительном значении запроса память отводится вплотную от вершины словаря, а при отрицательном значении запроса происходит освобождение соответствующего участка памяти. Наконец, слово , (запятая) выполняет так называемую «компиляцию» значения, снимаемого со стека: значение переносится на вершину словаря, после чего указатель вершины продвигается на 2 (размер в байтах скомпилированного значения). Некоторые из приведенных слов легко выражаются через другие:

```
: , ( A ---> ) HERE ! 2 ALLOT ;
: 2@ ( A ---> BB) DUP 2 + @ SWAP @ ;
```

Такая избыточность позволяет программисту быстрее находить нужные ему решения и делает программы более удобочитаемыми.

А как создать в словаре поименованную область памяти? Можно завести область и связать ее адрес с именем через описание константы: HERE 10 ALLOT CONSTANT X10 . Слово HERE оставляет на стеке адрес текущей вершины словаря, затем при исполнении текста 10 ALLOT от этой вершины резервируется 10 байт, после чего слово CONSTANT связывает адрес зарезервированной области с именем X10. В дальнейшем при исполнении слова X10 этот адрес будет положен на стек.

Другой возможный путь состоит в использовании слова CREATE (создать) → в таком контексте: CREATE X10 10 ALLOT . Слово CREATE, подобно слову VARIABLE , выбирает из входной строки очередное слово и определяет его как новую команду с таким действием: положить на стек адрес вершины словаря на момент создания этого слова. Поскольку следующие действия в приведенном примере резервируют память, то слово X10 будет класть на стек адрес зарезервированной области. Очевидно, что слово VARIABLE можно выразить через CREATE

```
: VARIABLE  ( ---> ) CREATE 2 ALLOT ;
```

или иначе (если мы хотим инициализировать нулем значение создаваемой переменной):

```
: VARIABLE  ( ---> ) CREATE 0 , ;
```

(Другой аспект использования слова CREATE рассматривается в п. 1.10).

В стандарте определен ряд системных переменных, к которым программист может свободно обращаться, в том числе STATE (состояние) → A и BASE (основание) → A . Исполнение каждого из этих слов заключается в том, что на стеке оставляется адрес ячейки, в которой хранится значение данной переменной.

Переменная STATE представляет текущее состояние текстового интерпретатора: нуль для исполнения и не нуль (обычно —1) для компиляции. Поэтому вся реализация слов [и] , переключающих интерпретатор из одного состояния в другое, сводится к одному присваиванию:

```
: [ ( ---> ) 0 STATE ! ; IMMEDIATE
: ] ( ---> ) -1 STATE ! ;
```

Переменная BASE хранит текущее основание системы счисления для ввода — вывода чисел, поэтому реализация слов для установки стандартных систем выглядит так:

```
: DECIMAL ( ---> ) 10 BASE ! ;
: HEX     ( ---> ) 16 BASE ! ;
```

Отсюда следует более изящный способ кратковременной смены системы счисления во время компиляции определения: [BASE @ HEX [FF00 [BASE !] . Сначала на стеке запоминается текущее основание, и система счисления переключается на основание 16, в котором и воспринимается следующее число FF00, после чего восстанавливается прежнее основание. А как узнать текущее основание системы счисления? Исполнение текста BASE @ не поможет, поскольку ответом всегда будет 10 (почему?). Правильный ответ даст исполнение текста BASE @ DECIMAL . , в результате чего значение основания будет напечатано как число в десятичной системе. Еще более правильным было бы

использовать текст BASE @ DUP DECIMAL . BASE !, который после печати основания в десятичной системе восстанавливает его прежнее значение.

1.6. Логические операции

В языке Форт имеется только один тип значений — 16-разрядные двоичные числа, которые, как мы видели, рассматриваются в зависимости от ситуации как целые числа со знаком или как адреса и т. д. Точно так же подходят и к проблеме представления логических значений ИСТИНА и ЛОЖЬ: число 0, в двоичном представлении которого все разряды нули, представляет значение ЛОЖЬ, а любое другое 16-разрядное значение понимается как ИСТИНА. Вместе с тем стандартные слова, которые должны возвращать в качестве результата логическое значение, из всех возможных представлений значения ИСТИНА используют только одно: число — 1 (или, что то же самое, 65535), в двоичном представлении которого все разряды единицы. Такое соглашение связано с тем, что традиционные логические операции конъюнкции, дизъюнкции и отрицания выполняются в Форте поразрядно над всеми шестнадцатью разрядами операндов:

```
AND    A,B ---> A   B    логическое И
OR     A,B ---> A   B    логическое ИЛИ
XOR    A,B ---> A   B    исключающее ИЛИ
NOT      A ---> ^ A      логическое НЕ
```

Как и в предыдущих случаях, эти операции не являются независимыми: операция отрицания (поразрядное инвертирование) легко выражается через исключающее ИЛИ (поразрядное сложение по модулю два):

```
: NOT  ( A ---> ^A )   -1 XOR ;
```

Нетрудно увидеть, что для принятого в Форте стандартного представления значений ИСТИНА и ЛОЖЬ все эти слова работают, как обычные логические операции.

Логические значения возникают в операциях сравнения, которые входят в обязательный набор слов и имеют общепринятую программистскую мнемонику:

```
<     A,B ---> A < B    меньше
=     A,B ---> A = B    равно
>     A,B ---> A > B    больше
```

Эти операции снимают со стека два врехних значения, сравнивают их как числа со знаком (операция «равно» выполняет поразрядное сравнение) и возвращают результат сравнения как значение ИСТИНА и ЛОЖЬ в описанном выше стандартном представлении. Из-за стремления к минимизации обязательного набора операций в него не включены слова для операций смешанного сравнения, поскольку их легко выразить через уже имеющиеся:

```
: <=  ( A,B ---> A <= B )  SWAP  <  NOT ;
: >=  ( A,B ---> A >= B )  SWAP  >  NOT ;
: <>  ( A,B ---> A <> B )  =  NOT ;
```

Для сравнения 16-разрядных чисел без знака имеется слово U< $A,B \to A < B$. Эта операция обычно используется для сравнения адресов, которые лежат в диапазоне от 0 до 65535. Буква U (от UNSIGNED — беззнаковый) в ее мнемонике говорит о том, что операнды рассматриваются как числа без знака.

Ввиду частого использования и возможности непосредственной реализации на многих существующих ЭВМ в обязательный набор слов включены одноместные операции сравнения с нулем:

```
0<    A ---> A < 0
0=    A ---> A = 0
0>    A ---> A > 0
```

При этом слово 0= можно использовать вместо NOT как операцию логического отрицания, и в отличие от NOT оно будет правильно работать при любых представлениях логического значения ИСТИНА.

Описанные выше двухместные операции сравнения естественным образом выражаются через сравнения с нулем:

```
: <  ( A,B ---> A < B )   -  0<  ;
: =  ( A,B ---> A = B )   -  0=  ;
: >  ( A,B ---> A > B )   -  0>  ;
```

Стандартное расширение двойных чисел имеет аналогичные слова для сравнения 32-разрядных значений:

```
D0=    AA ---> AA = 0
D<     AA,BB ---> AA < BB
D=     AA,BB ---> AA = BB
DU<    AA,BB ---> AA < BB
```

Слова D< и DU< различаются тем, что первое рассматривает свои операнды как числа со знаком, а второе — как числа без знака. Для слов D0= и D= такое различие несущественно, их, например, можно определить так:

```
: D0=  ( AA ---> AA = 0 )   OR  0=  ;
: D=   ( AA,BB ---> AA = BB )  D-  D0=  ;
```

Слово OR (логическое ИЛИ) в определении слова D0= логически складывает старшую и младшие половины исходного 32-разрядного значения. Нулевой результат будет получен тогда и только тогда, когда исходное значение было нулевым. Следующее слово 0= преобразует этот результат к логическому значению в стандартном представлении. Исполнение слова D= состоит в вычислении разности его операндов в сравнении этой разности с нулем.

1.7. Структуры управления

Во всех приводившихся выше определениях слов тело определения записывалось как последовательность уже известных слов-команд; семантика определяемого таким образом слова состоит в последовательном выполнении слов-команд тела. Помимо последовательного исполнения традиционными приемами в программировании стали *ветвление* (выбор между разными последовательностями действий) и *цикл* (многократное повторение одной последовательности действий).

В языке Форт тоже имеются условные операторы и циклы, реализованные с помощью специальных слов-команд, которые легко выражаются через другие, более элементарные слова (см. гл. 2). Это открывает путь к реализации таких вариантов структур управления, которые больше всего подходят для данной задачи,

что дает программисту широкие возможности для создания новых структур управления.

Стандарт языка предусматривает ряд слов для построения условных операторов и циклов в некотором конкретном виде. Эти слова используются внутри определений через двоеточие и разделяют тело определения на отрезки. Действия, соответствующие словам из этих отрезков, выполняются, не выполняются или выполняются многократно в зависимости от условий, проверяемых во время выполнения данного определения. Условные операторы и циклы могут свободно вкладываться друг в друга.

Условный оператор строится с помощью слов:

```
IF     A --->        (исполнение)
ELSE     --->        (исполнение)
THEN     --->        (исполнение)
```

Внутри определения через двоеточие отрезок текста IF (часть-то) ELSE (часть-иначе) THEN задает следующую последовательность действий. Слово IF (если) снимает значение с вершины стека и рассматривает его как логическое. Если это ИСТИНА (любое нулевое значение), то выполняется часть «то» — слова, находящиеся между IF и ELSE , а если ЛОЖЬ (равно нулю), то исполняется часть «иначе» — слова между ELSE и THEN . Сами слова ELSE (иначе) и THEN (то) играют роль ограничителей для слова IF и самостоятельной семантики не имеют. Часть «иначе» вместе со словом ELSE может отсутствовать, и тогда условный оператор имеет сокращенную форму IF (часть-то) THEN . Если логическое значение, снимаемое со стека словом IF, ИСТИНА, то выполняются слова, составляющие часть «то», а если ЛОЖЬ, то данный оператор не выполняет никаких действий. Обратите внимание, что условие для слова IF вычисляется предшествующими словами.

Для примера рассмотрим определение слова ABS, вычисляющего абсолютное значение числа:

```
: ABS ( A --->абс A) DUP 0< IF NEGATE THEN ;
```

Слово DUP дублирует исходное значение, следующее слово 0< проверяет его знак, заменяя копию на логическое значение — результат проверки. Слово IF снимает со стека этот результат, и если это ИСТИНА,

то лежащее на стеке исходное отрицательное значение словом NEGATE заменяется на противоположное.

Еще пример: стандартное слово ?DUP дублирует верхнее значение, если это не ноль, и оставляет стек в исходном состоянии, если на вершине ноль:

```
: ?DUP  ( A ---> A,A/0 ) DUP  IF DUP THEN  ;
```

В спецификации данного слова косая черта разделяет два варианта результата, который это слово может оставить на стеке. Примером использования полного условного оператора может служить определение слова S > D , расширяющего исходное значение до значения двойной длины распространением знакового разряда:

```
: S>D   ( A ---> AA )
            DUP  0<  IF  -1  ELSE  0  THEN  ;
```

Это слово добавляет в стек — 1 , если число на вершине стека отрицательно, или 0 в противном случае. Таким образом, выполняется распространение знакового разряда на старшую половину значения двойной точности.

В стандарт языка Форт включены *циклы с условием* и *циклы со счетчиком*. Циклы первой группы образуются с помощью слов

```
BEGIN       --->        (исполнение)
UNTIL     A --->        (исполнение)
WHILE     A --->        (исполнение)
REPEAT      --->        (исполнение)
```

и имеют две формы:

```
BEGIN  <тело>  UNTIL
BEGIN  <тело-1>  WHILE  <тело-2>  REPEAT
```

В обоих случаях цикл начинается словом BEGIN (начать), которое служит открывающей скобкой, отмечающей начало цикла, и во время счета не выполняет никаких действий.

Цикл BEGIN—UNTIL называется циклом с *проверкой в конце*. После исполнения слов, составляющих его тело, на стеке остается логическое значение — условие завершения цикла. Слово UNTIL (пока не) снимает это значение со стека и анализирует его. Если это ИСТИНА (не нуль), то исполнение цикла завершается, т. е. далее будут исполняться слова, следующие за UNTIL,

а если это ЛОЖЬ (нуль), то управление возвращается к началу цикла от слова BEGIN . Например, определение слова, вычисляющего факториал, может выглядеть так:

```
: ФАКТОРИАЛ ( N ---> N!  ВЫЧИСЛЕНИЕ N ФАКТОРИАЛ)
   DUP 2 < IF DROP 1   ( 1     ЕСЛИ N<2, ТО N!=1)
          ELSE         ( N     ИНАЧЕ            )
     DUP               ( S,K      S=N,K=N       )
    BEGIN              ( S,K                    )
        1-             ( S,K'     K'=K-1        )
          SWAP  OVER   ( K',S,K'                )
          *  SWAP      ( S',K'    S'=S*K'       )
        DUP  1  =      ( S',K',K'=1             )
    UNTIL              ( S',1     S'=N!         )
  DROP  THEN  ;        ( N!                     )
```

Как и ранее, в комментариях, сопровождающих каждую строчку текста, мы указываем значения, которые остаются на вершине стека после ее исполнения. Такое документирование облегчает понимание программы и помогает при ее отладке.

Цикл *с проверкой в начале* BEGIN — WHILE — REPEAT используется, когда в цикле есть действия, которые не надо выполнять в заключительной итерации. Исполнение слов, составляющих его тело-1, оставляет на стеке логическое значение — условие продолжения цикла. Слово WHILE (пока) снимает это значение со стека и анализирует его. Если это ИСТИНА (не нуль), то исполняются слова, составляющие тело-2 данного цикла до ограничивающего слова REPEAT (повторять), после чего вновь исполняется тело-1 от слова BEGIN. Если же значение условия ЛОЖЬ (нуль), то исполнение данного цикла завершается и начинают выполняться слова, следующие за REPEAT . Заметьте, что в отличие от цикла BEGIN-UNTIL, значение ИСТИНА соответствует продолжению цикла. Для примера рассмотрим программу вычисления наибольшего общего делителя по алгоритму Евклида:

```
НОД   ( A,B ---> C:НАИБОЛЬШИЙ ОБЩИЙ ДЕЛИТЕЛЬ)
2DUP < IF SWAP THEN    ( ТЕПЕРЬ  A>=B   )
   BEGIN DUP           ( A,B,B )
   WHILE               ( ПОКА B НЕ НОЛЬ )
         2DUP  MOD     ( A,B,C:ОСТАТОК  )
```

```
      ROT                  ( B,C,A  )
      DROP                 ( A',B'   A'=B,B'=C)
REPEAT   DROP  ;           ( НОД)
```

Для организации циклов с целочисленной переменной — *счетчиком цикла* — используются слова

```
DO      A,B --->      (исполнение)
LOOP        --->      (исполнение)
+LOOP     A --->      (исполнение)
I           ---> A    (исполнение)
J           ---> A    (исполнение)
LEAVE       --->      (исполнение)
```

Такие циклы записываются в одной из следующих двух форм: DO <тело> LOOP или DO<тело> +LOOP. В обоих случаях цикл начинается словом DO (делать), которое снимает со стека два значения: начальное (на вершине стека) и конечное (второе сверху) — и запоминает их. Текущее значение счетчика полагается равным начальному значению, после чего исполняются слова, составляющие тело цикла. Слово LOOP увеличивает текущее значение счетчика на единицу и проверяет условие завершения цикла. В отличие от него, слово + LOOP прибавляет к текущему значению счетчика значение шага, которое вычисляется на стеке телом цикла и рассматривается как число со знаком. В обоих случаях условием завершения цикла является пересечение границы между A—1 и A при переходе от прежнего значения счетчика к новому (направление перехода определяется знаком шага), где A — конечное значение, снятое со стека словом DO.

Если пересечения не произошло, то тело цикла исполняется вновь с новым значением счетчика в качестве текущего.

Такое определение позволяет рассматривать исходные параметры цикла (начальное и конечное значения) и как числа со знаком, и как числа без знака (адреса). Например, текст 10 O DO (тело) LOOP предписывает выполнять тело цикла 10 раз со значениями счетчика 0, 1, 2, ..., 9, а в случае 0 10 DO (тело) LOOP тело цикла будет исполнено 65 526 раз со значением счетчика 10, 11, ..., 32 767, -32 768, -32 767, ..., -1 или (что то же самое) со значениями счетчика 10, 11, ..., 65 535.

В то же время цикл 0 10 DO (тело) — 1 +LOOP будет исполняться 11 раз (а не 10) со значениями счетчика 10, 9, ..., 0, поскольку пересечение границы между —1 и 0 произойдет при переходе от значения счетчика 0 к следующему значению — 1. Цикл 10 0 DO (тело) — 1 + LOOP будет исполняться 65 527 раз со значениями счетчика 0, —1, —2, ..., —32 768, 32 767, ... 10 или (что то же самое) 0, 65 535, 65 534. . . . .10.
Таким образом, цикл со счетчиком всегда выполняется хотя бы один раз. Некоторые реализации предусматривают слово ?DO , которое не исполняет тело цикла ни разу, если начальное и граничное значения оказались одинаковыми.

Внутри тела цикла слово I кладет на стек текущее значение счетчика. Например, следующее определение вычисляет сумму квадратов первых *n* натуральных чисел:

```
: SS2   ( N ---> S:СУММА КВАДРАТОВ ОТ 1 ДО N)
   0  SWAP            ( 0,N          S[0]=0 )
   1+  1              ( S[0],N+1,1          )
        DO   I        ( S[I-1],I            )
          DUP  *  +   ( S[I] S[I]=S[I-1]+I*I)
        LOOP ;        ( S[N]   )
```

Слово LEAVE (уйти), употребленное внутри цикла, вызывает прекращение исполнения его тела; управление переходит к словам следующим за словом LOOP или + LOOP .

В программировании часто применяется конструкция *цикл в цикле*. Чтобы во внутреннем цикле получить текущее значение счетчика объемлющего цикла, используется слово J: DO ... I ... DO ... I ... J ... LOOP ... I ... LOOP . Первое вхождение слова I дает текущее значение счетчика внешнего цикла. Следующее вхождение I дает уже значение счетчика внутреннего цикла. Чтобы получить счетчик внешнего цикла, надо использовать слово J.

По выходе из внутреннего цикла доступ к этому значению вновь дает слово I. Этим слова I и J отличаются от переменных цикла в традиционных языках программирования. Некоторые реализации предусматривают еще и слово K для досступа к счетчику третьего объемлющего цикла.

1.8. Литеры и строки, форматный вывод чисел

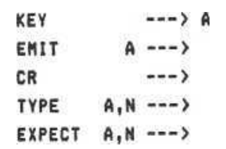
В современном программировании важное место занимает обработка текстовых данных. С каждой литерой, которую можно ввести с внешнего устройства или вывести на него, связывается некоторое число — код этой литеры, так что в памяти ЭВМ литеры представлены своими кодами. Стандарт языка Форт предусматривает использование таблицы кодов ASCII, в которой задействованы все числа в диапазоне от 0 до 127. Каждый код занимает один 8-разрядный байт, в котором для представления литеры используются младшие 7 разрядов.

Такая привязка к одной конкретной кодировке не является существенным препятствием к использованию других, если сохраняется условие, что код литеры занимает один байт. В применяемых в нашей стране форт-системах помимо ASCII применяются коды КОИ-7, КОИ-8, ДКОИ и другие.

Для доступа к однобайтным значениям, расположенным в памяти, используются слова C@ A → **B** **и C! B,A** → , которые аналогичны словам @ и ! . Префикс C (от слова CHARACTER — литера) говорит о том, что эти слова работают с литерными кодами (однобайтными значениями).

Слово C@ возвращает содержимое байта по адресу A, дополняя его до 16-разрядного значения нулевыми разрядами. Обратное действие выполняет слово C! , которое засылает младшие 8 разрядов значения B в память по адресу A.

Для политерного обмена с терминалом стандарт предусматривает такие слова:



Слово KEY (клавиша) возвращает в младших разрядах значения A — код очередной литеры, введенной с терминала. В отличие от слова C@ старшие разряды зависят от реализации и могут быть ненулевыми. Обратное действие выполняет слово EMIT (испустить), которое снимает значение со стека и, рассматривая его

2 Зак. № 966

младшие разряды как код литеры, выводит эту литеру на терминал. Специальное слово CR (сокращение от CARRIAGE RETURN — возврат каретки) выполняет перевод строки при выводе на терминал.

Расположенные в памяти побайтно коды литер образуют строку, которую можно напечатать на терминале словом TYPE (напечатать). Это слово снимает со стека число — количество литер (оно должно быть неотрицательно) и адрес начала строки (ее первого байта):

```
: TYPE  ( A,N ---> )       ?DUP  IF  0  DO
     DUP  I  +  C@  EMIT  LOOP  THEN   DROP ;
```

Если число литер равно нулю, то ничего не печатается.

Обратное действие — ввод строки литер — выполняет слово EXPECT (ожидать), которое снимает со стека длину и адрес области памяти для размещения вводимых литер. Коды литер, последовательно вводимых с терминала, помещаются в указанную область до тех пор, пока не будет введено заданное число литер или не будет введена управляющая литера «возврат каретки» (код этой литеры в память не заносится). Фактическое число введенных литер сообщается в стандартной переменной SPAN (размер), эти литеры к тому же отображаются на терминале.

Ввиду его особой важности для кодирования пробела выделена специальная константа BL (от BLANK — пробел), которую для кода ASGII можно задать так: 32 CONSTANT BL. При исполнении слова BL на стеке остается код пробела. Чтобы вывести пробел на терминал, имеются следующие стандартные слова:

```
: SPACE  (  ---> )   BL  EMIT ;
: SPACES (  N--->) ?DUP IF 0 DO SPACE LOOP THEN ;
```

Слово SPACE (пробел) выводит на терминал один пробел, а слово SPACES (пробелы) — несколько, снимая их количество со стека (это значение, как и длина строки в слове TYPE, должно быть неотрицательным).

Внутри определений через двоеточие можно использовать явно заданные тексты для вывода их на терминал. При исполнении слова ." (точка и кавычка),

употребленного в контексте ." текст ", следующие за ним литеры до закрывающей кавычки исключительно печатаются на терминале. Пробел, отделяющий слово .", в число печатаемых литер не входит. Другое слово .((точка и скобка) отличается от ." тем, что ограничителем текста является закрывающая правая скобка, и это слово можно использовать как внутри определений, так и вне их.

Помимо строки — поля байт, длина которого задается отдельно — язык Форт использует *строки со счетчиком.* Строка со счетчиком представляется полем байт, причем в первом байте записана длина строки. Стандарт не определяет форму представления счетчика длины, оставляя решение этого вопроса на усмотрение разработчиков конкретной реализации. Для перехода от строки со счетчиком к строке с явно заданной длиной имеется слово COUNT (счетчик) *A* → B,N, которое преобразует адрес A строки со счетчиком в адрес B ее первой литеры (обычно это A+ 1) и значение счетчика. Строки со счетчиком используются при вводе слов из входной строки. Стандартное слово WORD (слово) C→ A снимает со стека код литеры-ограничителя и выделяет из входной строки подстроку, ограниченную этой литерой (начальные вхождения литеры-ограничителя пропускаются). Из выделенной подстроки формируется строка со счетчиком, адрес которой возвращается в качестве результата. Слова-команды языка Форт вводятся исполнением текста BL WORD, а текстовая строка в слове ." — исполнением текста QUOTE WORD , где слово QUOTE — константа, обозначающая код кавычки. Литеры введенной строки обычно располагаются вслед за вершиной словаря, т. е. в незащищенном месте, и поэтому их нужно как-то защитить, если предполагается их дальнейшее использование. Некоторые реализации предусматривают слово" (кавычка), которое используется внутри определения через двоеточие и во время его исполнения кладет на стек адрес следующей строки как строки со счетчиком. Это позволяет работать с явно заданными текстами.

Чтобы программист мог задавать коды литер, не связывая себя конкретной кодировкой, во многие реализации введено слово C" , которое кладет на стек код первой литеры следующего слова и может исполь-

зоваться как внутри определения через двоеточие, так и вне его:

```
: C" ( ---> C )  BL WORD COUNT DROP
        C@ [COMPILE] LITERAL ; IMMEDIATE
```

Исполнение текста BL WORD COUNT DROP C@ оставляет на стеке код первой литеры следующего слова. Далее этот код нужно либо скомпилировать как число, либо оставить на стеке в зависимости от текущего состояния текстового интерпретатора. Для этого используется уже известное нам слово LITERAL . Однако включить его непосредственно в текст нельзя, так как это слово имеет признак немедленного исполнения и будет исполняться во время компиляции данного определения. Чтобы скомпилировать слово с признаком немедленного исполнения, используется слово [COMPILE] (от COMPILE — компилировать) → (компиляция), которое само имеет такой признак. Оно принудительным образом компилирует следующее за ним слово независимо от наличия у него признака немедленного исполнения. Таким образом, ввод строки, ограниченной кавычкой, с помощью слова C" можно задать так: C" " WORD . Такой текст более нагляден, чем тот, в котором используется конкретный код или обозначающая его константа.

Важной областью применения строковых значений являются форматные преобразования, позволяющие переводить число из машинного двоичного представления в строку литер. Эти преобразования выполняются над числами двойной длины, результирующая строка размещается во временном буфере PAD (прокладка), который заполняется с конца. Такое название буфера связано с тем, что он располагается в незащищенной части адресного пространства между словарем и стеком. Слово PAD кладет на стек адрес конца этого буфера и обычно определяется так:

```
: PAD ( ---> A )  HERE 100 + ;
```

В данном случае предполагается, что размер буфера не будет превышать 100 байт.

Собственно форматное преобразование начинается словом <# , которое устанавливает служебную переменную HLD на конец буфера PAD:

```
: <#  ( ---> )  PAD HLD ! ;
```

Занесение очередной литеры в буфер PAD выполняет слово HOLD (сохранить):

```
: HOLD  ( C ---> )  -1 HLD +!   HLD @ C! ;
```

Преобразование числа выполняет слово # DD1 ⟶ DD2, которое работает со значениями двойной длины. Параметр делится на текущее значение переменной BASE (основание системы счисления) и заменяется на стеке получившимся частным, а остаток переводится в литеру соответствующую ему как цифра в данной системе счисления, и через слово HOLD эта литера добавляется в буфер PAD. Полный перевод числа выполняет слово #S:

```
: #S  ( DD --->0,0)  BEGIN # 2DUP D0= UNTIL ;
```

которое выделяет цифры числа последовательным делением, пока не получится нуль. Наконец, слово #> завершает форматное преобразование, возвращая адрес и длину получившейся текстовой строки:

```
: #>  ( DD ---> A,N) 2DROP  HLD @  PAD OVER  - ;
```

Для вывода знака «минус» имеется слово SIGN (знак):

```
: SIGN  (  A ---> )  0<  IF  C" -  HOLD  THEN  ;
```

которое добавляет в.буфер PAD знак «минус», если параметр на вершине стека (число одинарной точности) отрицателен.

С помощью перечисленных средств легко определить стандартные слова D. и . для печати чисел и в свободном (минимальном) формате:

```
: D.  (  DD ---> )
   2DUP  DABS       (  DD,DDABS )
    <#  #S          (  DD,0,0  )
        ROT         (  D-МЛ,0,0,D-СТ )
        SIGN        (  D-МЛ,0,0  )
      #>            (  D-ML,A,N  )
   TYPE  SPACE  DROP ;
: .  ( N ---> )  S>D  D. ;
```

Слово D. сначала переводит абсолютное значение исходного числа в строку литер, потом добавляет к ней возможный знак «минус», анализируя для этого стар-

шую половину первоначального значения, и затем печатает получившуюся строку, выводя после нее еще один пробел. Слово . дополняет свой параметр до значения двойной длины распространением знакового разряда и обращается к слову D. для печати получившегося числа. Аналогичным образом реализуются стандартные слова D.R и .R , которые печатают число в поле заданного размера вплотную к его правому краю (отсюда в их мнемонике присутствует буква R от RIGHT — правый), добавляя при необходимости начальные пробелы:

```
: D.R  ( DD:ЧИСЛО,F:РАЗМЕР ПОЛЯ ---> )
   OVER  2SWAP  DABS        ( D-CT,F,DD,DDABS)
    <#  #S  ROT  SIGN  #>   ( F,A,N )
   ROT  OVER  -             ( A,N,F-A )
   DUP  0>  IF SPACES ELSE DROP THEN  TYPE ;
: .R   ( N:ЧИСЛО,F:РАЗМЕР ПОЛЯ ---> )
   SWAP  S>D  D.R ;
```

В заключение рассмотрим программу шестнадцатиричной распечатки областей памяти словом DUMP (дамп), которое получает на стеке адрес области и ее длину:

```
: DUMP   ( A:АДРЕС,N:ДЛИНА ---> )
   OVER  BASE  @  HEX  2SWAP      ( A,B,A,N)
   +  ROT  -2  AND                ( B,A+N,A)
   DO  I  <#  C" #  HOLD          ( B,AI )
     0 15 DO  DUP  I  +           ( B,AI,AI+J)
      C@  DECODE  HOLD  -1 +LOOP  ( B,AI )
             C" #  HOLD           ( B,AI )
     0 14 DO  BL HOLD  DUP I +    ( B,AI,AI+J)
      @ 0  # #  2DROP  -2 +LOOP   ( B,AI )
  BL HOLD  BL HOLD  0  # # # # #> ( B,AT,NT)
         CR TYPE  16 +LOOP   BASE !  ;
```

Внешний цикл с шагом 16 формирует и печатает текстовую строку. Первый внутренний цикл с шагом —1 засылает в буфер PAD литерные значения, соответствующие распечатываемым байтам. Здесь слово DECODE C → C/C1 заменяет код C на некоторый код C1 (например, код литеры «точка»), если он не является кодом литеры, которую можно напечатать на данном терминале. Второй внутренний цикл с шагом —2 засылает в буфер четыре шестнадцатиричные цифры —

представления двухбайтных значений, разделяя их одним пробелом. Далее в буфере PAD (т. е. в начале строки) формируется адрес тоже в виде четырехзначного числа. Перед началом работы устанавливается шестнадцатиричная система счисления, а в конце восстанавливается первоначальная. Следующий протокол работы показывает результат исполнения слова DUMP в конкретном случае:

```
> 1000 40 DUMP
03E8  03C0 03EC 45E0 A220 0003 4520 A390 07FA *...X..S.....TM.3*
03F8  04C5 D4C9 E300 03E4 0402 9180 A331 4780 *.EMIT..U..JLT..U*
0408  A414 45E0 A230 0004 4520 A37A 9180 A330 *U...S.....T:JLT.* OK
```

Обратите внимание, что адреса и значения байтов напечатаны в шестнадцатиричной системе. Точки в литерном представлении байтов заменяют литеры, которые не могут быть напечатаны.

1.9. Определяющие слова

Конструкции языка Форт, которые мы рассматривали до сих пор, имеют аналоги в других известных языках программирования. Например, определению через двоеточие очевидным образом соответствует описание процедуры в языках Фортран и Паскаль. Теперь мы рассмотрим свойство языка Форт, которое существенно отличает его от других и в значительной степени определяет его как нечто новое.

Программируя какую-либо задачу, мы моделируем понятия, в которых эта задача формулируется и решается, через понятия данного языка программирования. Традиционные языки имеют определенный набор понятий (процедуры, переменные, массивы, типы данных, исключительные ситуации и т.д.), с помощью которых программист должен выразить решение исходной задачи. Этот набор выразительных средств фиксирован в каждом языке, но поскольку он содержит такие универсальные средства, как процедуры и типы данных, то опытный программист может смоделировать любые необходимые ему понятия программирования.

Язык Форт предлагает вместо фиксированного набора порождающих понятий единый механизм порождения таких порожающих понятий. Таким образом, приступая к программированию задачи на языке Форт,

программист определяет необходимые ему понятия и с их помощью решает поставленную задачу. Такое прямое введение в язык нужных для данной задачи средств вместо их моделирования (часто весьма сложного) через имеющиеся универсальные средства уменьшает разрыв между постановкой задачи и ее программной реализацией. Это упрощает понимание и отладку программы, так как в ней более наглядно проступают объекты и отношения исходной задачи, и вместе с тем повышает ее эффективность, так как вместо сложного моделирования понятий задачи через универсальные используется их непосредственная реализация через элементарные.

Новые понятия вводятся посредством определения через двоеточие, в теле которого используются слова

```
CREATE     --->
DOES>      --->       (компиляция)
           ---> A     (исполнение)
```

Рассмотрим уже известное нам слово CONSTANT (константа), которое используется для определения констант. Его определение можно задать так:

```
: CONSTANT ( N ---> )  CREATE , DOES> @ ;
```

Часть определения от слова CREATE до DOES> называется *создающей* (CREATE — создать), остальная часть от слова DOES> и до конца называется *исполняющей* (DOES — исполняет). В данном случае создающая часть состоит из одного слова , (запятая), а исполняющая часть — из слова @ (разыменование).

Рассмотрим исполнение данного определения на примере 4 CONSTANT ХОР. Слово 4 кладет число 4 на стек. Далее исполняется слово CONSTANT. Слово CREATE , с которого начинается его определение, выбирает из входной строки очередное слово (в данном случае ХОР) и добавляет его в словарь как новую команду. Создающая часть, состоящая из слова «запятая», переносит число 4 в память, компилируя его на вершину словаря. Слово DOES> , отмечающее конец создающей части, завершает исполнение данного определения, при этом семантикой созданного слова ХОР будет последовательность действий исполняющей части, начиная от слова DOES> . В дальнейшем исполнение слова ХОР начнется с того, что слово

DOES> положит на стек адрес вершины словаря, какой она была на момент начала работы создающей части, после чего будет работать исполняющая часть определения. Поскольку по данному адресу создающая часть скомпилировала число 4, то исполняющая часть — разыменование — заменит на стеке этот адрес его содержимым, т. е. числом 4, что и требуется по смыслу данного понятия.

Рассмотрим другой пример. Введем понятие вектора. При создании вектора будем указывать размер (число элементов), а при обращении к нему — индекс (номер) элемента, в результате чего получается адрес данного элемента. Этот адрес можно разыменовать и получить значение элемента или можно заслать по этому адресу новое значение. Если желательно контролировать правильность индекса при обращении к вектору, то определение может выглядеть так:

```
: ВЕКТОР ( N:РАЗМЕР--->) CREATE DUP ,  2* ALLOT
   DOES>  ( I:ИНДЕКС,A--->A[I]:АДРЕС ЭЛ-ТА I)
     OVER  1- OVER @ U<  ( ПРОВЕРКА ИНДЕКСА)
       IF  SWAP 2*  +  EXIT THEN
        ." ОШИБКА В ИНДЕКСЕ"  ABORT  ;
```

Разберем, как работает данное определение при создании вектора 10 вектор X.

Создающая часть компилирует размер вектора и вслед за этим отводит память на 10*2, т. е. 20 байт. Таким образом, для вектора X в словаре отводится область размером 22 байта, в первых двух байтах которой хранится число 10 — размер вектора. При обращении к вектору X на стеке должно находиться значение индекса. Слово DOES> кладет сверху адрес области, сформированной создающей частью, после чего работает исполняющая часть определения. Проверив, что индекс I лежит в диапазоне от 1 до 10, она оставляет на стеке адрес, равный начальному адресу области плюс 1*2, т. е. адрес I-го элемента вектора, если считать, что элементы располагаются в зарезервированной области подряд. Слово EXIT (выход) завершает исполнение определения, что позволяет обойтись без части «иначе» в условном операторе. Если окажется, что индекс не положителен или больше числа элементов, то будет напечатано сообщение "ошибка в индексе" словом ." , и исполнение закончится через слово ABORT

(выброс). Если по каким-либо причинам контроль индексов не нужен, можно дать более краткое определение:

```
: ВЕКТОР  ( N:РАЗМЕР ---> )  CREATE  2 *  ALLOT
    DOES>  ( I:ИНДЕКС,A ---> A[I]:АДРЕС ЭЛ-ТА I)
              SWAP  1-  2 *  +  ;
```

Если мы условимся считать индексы не от единицы, а то нуля, то исполняющая часть еще более сократится за счет исключения слова I — для уменьшения значения индекса на единицу.

Таким образом, программист может реализовывать варианты понятий, наиболее подходящие для его задачи. Исходный небольшой набор слов-команд форт-системы он может избирательно наращивать в нужном направлении, постоянно совершенствуя свой инструментарий.

Используемый в языке Форт способ введения определяющих слов связан с очень важным понятием — *частичной параметризацией*. Определяющее слово задает целый класс слов со сходным действием, которое описывается исполняющей частью определяющего слова. Каждое отдельное слово из данного класса характеризуется результатом исполнения создающей части — тем или иным содержимым связанной с ним области памяти, адрес которой передается исполняющей части как параметр. Таким образом, исполняющая часть — то общее, что характеризует данный класс слов,— во время ее исполнения частично параметризуется результатом исполнения создающей части для данного отдельного представителя этого класса. Как создающая часть, так и частично параметризованная исполняющая часть, могут требовать дополнительных параметров для своего исполнения (в примере для вектора это размер вектора и индекс). Все это представляет программисту практически неограниченную свободу в создании новых понятий и удобных инструментальных средств.

Глава 2. **РЕАЛИЗАЦИЯ И РАСШИРЕНИЯ**

2.1. Шитый код и его разновидности

Логически можно выделить два подхода к реализации языков программирования — трансляцию и интерпретацию [10]. *Транслятор* преобразует входной текст программы в машинный код данной ЭВМ; впоследствии этот код, объединяясь с другими машинными модулями, образует рабочую программу, которую можно загрузить в оперативную память и исполнить. *Интерпретатор* непосредственно исполняет программу на языке высокого уровня, рассматривая входной текст как последовательность кодов операций, управляющих его работой. Между этими полюсами располагается целый спектр промежуточных подходов, состоящих в предварительном преобразовании входного текста программы в некоторой промежуточный язык с последующей интерпретацией получившегося кода. Чем ближе промежуточный язык к машинному языку, тем ближе данный подход к классической трансляции, а чем ближе он к исходному языку программирования, тем ближе этот подход к интерпретации.

Оптимальный вариант промежуточного языка должен существенно отличаться от исходного языка программирования и быть удобным для создания простых и надежных интерпретаторов. Примером одного из таких промежуточных языков является известный П-код, используемый во многих реализациях языка Паскаль. Рассматриваемые нами варианты шитого кода образуют специальный класс представлений промежуточных языков, особенно удобных для интерпретации [22, 25, 29].

В общем случае промежуточный язык состоит из набора элементарных операций, средств управления локальной памятью для хранения и передачи данных и средств управления процессом вычисления. **Однако** если по своей сложности промежуточный **язык** при-

ближается к исходному машиннонезависимому языку программирования, то целесообразность его использования снижается. Одним из приемов, направленных на упрощение набора команд промежуточного языка, является применение *стековой архитектуры* (или *нуль-адресных команд)*. В этом случае для работы с памятью требуется всего две операции: перенести значение со стека в память и наоборот из памяти в стек. Кроме того, стековая архитектура естественным образом приводит к стековому механизму передачи параметров, который таким образом становится общей частью всех реализаций данного промежуточного языка.

Шитый код особенно удобен для реализации виртуальных машин со стековой архитектурой: П-кода, Лиспа и, конечно, Форта. Вместе с тем в каждом случае следует различать качества реализации, которые обеспечиваются применением шитого кода, и качества, которые присущи самому языку.

Известны четыре разновидности шитого кода: подпрограммная, косвенная и свернутая. Во всех четырех случаях операции промежуточного языка представляются ссылками на подпрограммы, соответствующие этим операциям. Перечисленные разновидности различаются способом представления этих ссылок и соответственно интерпретатором такой последовательности. Подпрограммы, реализующие операции языка, разделяются на два класса: верхнего уровня, представленные в том же шитом коде, и нижнего уровня, реализованные непосредственно в машинном коде данной ЭВМ. Подпрограммы нижнего уровня взаимодействуют с интерпретатором шитого кода, используя выделенные ему машинные ресурсы. Вместе с тем конкретный вид этих ресурсов может быть самым разным для разных архитектур ЭВМ.

Во всех разновидностях шитого кода его интерпретатор должен обеспечивать выполнение трех действий, которые традиционно обозначаются так:

NEXT (следующий) — переход к интерпретации следующей ссылки в данной последовательности ссылок;

CALL (вызов) — переход в подпрограмму верхнего уровня, представленную в шитом коде;

RETURN (возврат) — возврат из подпрограммы верхнего уровня на продолжение интерпретации.

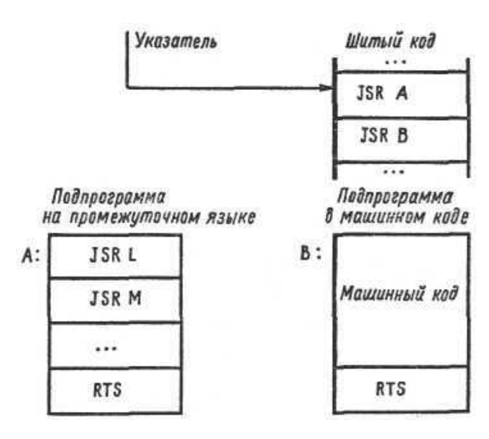


Рис. 2.1. Подпрограммный шитый код

В силу его очевидной рекурсивности интерпретатор использует специальный стек в качестве собственной структуры данных. Этот стек называется *стеком возвратов,* чтобы отличать его от *стека данных,* обычно используемого операциями промежуточного языка. В качестве еще одной собственной структуры данных интерпретатора выступает указатель на текущее место в интерпретируемой последовательности ссылок, представляющих операции промежуточного языка.

Из всех разновидностей шитого кода *подпрограммный* максимально эффективен по времени исполнения. Он удобен в том случае, когда архитектура данной ЭВМ включает аппаратные стеки, команду перехода на подпрограмму (перехода с возвратом), в которой адрес возврата запоминается на вершине стека, и команду возврата по адресу, находящемуся на вершине стека. Для ЭВМ СМ-4 и «Электроника-60» это команды JSR и RST, для микропроцессора К580 — команды CALL и RET.

Структура подпрограммного шитого кода приведена на рис. 2.1. Каждая ссылка на операцию промежуточного языка представляется в виде машинной команды перехода на соответствующую подпрограмму. Стек возвратов используется для сохранения адреса возврата этими командами, а в качестве указателя текущего места в интерпретируемой последовательности ссылок выступает внутренний регистр счетчика адреса. Высокоуровневая подпрограмма на промежуточном

языке представляет собой последовательность таких же команд перехода с возвратом, которая заканчивается командой возврата по адресу, снимаемому с вершины стека возвратов. Подпрограмма нижнего уровня должна заканчиваться такой же командой возврата для продолжения обработки. Таким образом, в подпрограммном шитом коде интерпретатор реализуется непосредственным образом в структуре самого кода. Действие NEXT состоит в исполнении пары команд JSR/RST, действие CALL как таковое отсутствует, действие RETURN состоит в команде RST. Подпрограммный шитый код в отличие от всех остальных разновидностей допускает большую оптимизацию по времени счета за счет непосредственной вставки подпрограмм нижнего уровня в место их вызова.

Прямой шитый код (рис.2.2.) уступает подпрограммному по скорости исполнения, но дает выигрыш по объему памяти, необходимой для его размещения. В качестве последовательности операций промежуточного языка выступает последовательность адресов соответствующих подпрограмм. Ее можно рассматривать как последовательность вызовов подпрограммного шитого кода с удаленным кодом команды JSR (именно это и дает экономию объема памяти примерно на 1/3 по сравнению с подпрограммным кодом). Поскольку код команды отсутствует, требуется специальный интер-

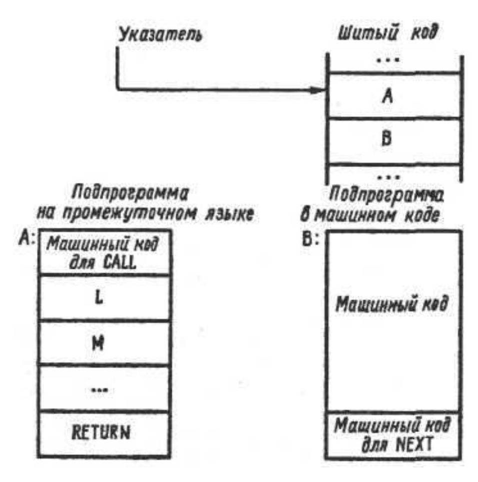


Рис. 2.2. Прямой шитый код

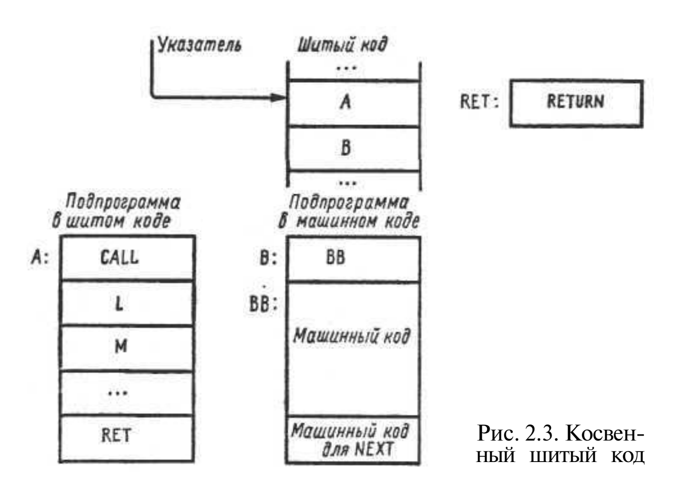


Рис. 2.3. Косвенный шитый код

претатор последовательности ссылок. Подпрограммы верхнего уровня должны начинаться машинными командами, выполняющими действие CALL (положить текущее значение указателя на стек возвратов, перевести указатель на начало последовательности адресов данной подпрограммы, исполнить NEXT) и заканчиваться адресом подпрограммы RETURN (снять значение со стека возвратов и заслать его в указатель, исполнить NEXT). Подпрограммы в машинном коде должны заканчиваться исполнением действия NEXT (скопировать адрес подпрограммы по текущему значению указателя в рабочую ячейку, перевести указатель на следующий элемент кода, передать управление по адресу в рабочей ячейке). В тех случаях, когда архитектура ЭВМ позволяет выразить действия CALL и NEXT одной-двумя машинными командами (например, для ЭВМ СМ-4), эти команды вставляются непосредственно в подпрограммы; если же требуется более длинная последовательность команд, то в подпрограммы вставляются команды перехода на соответствующие точки, которые выносятся в отдельное подпрограммное ядро.

Косвенный шитый код уступает прямому по скорости исполнения, но имеет то преимущество, что его высокоуровневые подпрограммы не зависят от машины, поскольку не содержат машинных кодов. Как и в случае прямого кода, последовательность операций промежуточного языка состоит из последовательности адресов подпрограмм, разница заключается в организации этих

подпрограмм и действиях интерпретатора (рис. 2.3.). Теперь чтобы передать управление на машинный код, в действии NEXT требуется выполнить еще одно разыменование. Подпрограмма верхнего уровня начинается не машинными командами для действия CALL, а ячейкой, где записан адрес этой точки, и заканчивается ячейкой, где записан адрес ячейки с адресом точки RETURN. Подпрограмму на машинном языке представляет ячейка, где записан адрес начала соответствующего кода. Завершающим действием такой подпрограммы, как и раньше, является исполнение действия NEXT. Для этого обычно используется безусловный переход на соответствующую точку интерпретатора.

Наконец, *свернутый шитый код,* который может быть как прямым, так и косвенным, отличается тем, что вместо прямых адресов подпрограмм и кодов в нем используются их свертки, которые, вообще говоря, короче этих адресов. Таким путем за счет усложнения доступа к подпрограммам в действии NEXT (через использование таблицы сверток или специальной функции, преобразующей свертку в соответствующий адрес) можно добиться экономии памяти или возможности использовать больший диапазон адресов для размещения кода. Пусть, например, архитектура ЭВМ требует, чтобы адреса подпрограмм и команд были четными. Тогда, используя в качестве свертки, представляющей такой адрес, его значение, деленное на 2, мы получаем возможность использовать вдвое большее адресное пространство.

Таким образом, разновидности шитого кода предлагают широкий диапазон характеристик по скорости исполнения и требованиям к памяти. Поскольку все они логически эквивалентны, то выбор конкретного варианта определяется конкретными требованиями. В реализациях языка Форт встречаются все эти варианты.

2.2. Структура словарной статьи

Каждое слово-команда, известное форт-системе, хранится в оперативной памяти (в словаре) в виде специальной структуры данных — *словарной статьи,* состоящей из поля имени, поля связи, поля кода и поля

параметров. Стандарт не фиксирует порядок взаимного расположения этих полей, оставляя его на усмотрение разработчиков реализации. Как правило, поля располагаются подряд в том порядке, как они перечислены выше.

Поля имени и связи вместе образуют *заголовок* словарной статьи, который нужен для поиска статьи по имени слова. Результатом поиска являются поля кода и параметров, которые вместе образуют собственно *тело* словарной статьи, определяющее действие, связанное с данным словом, т. е. его семантику. Согласно стандарту словарную статью представляет адрес ее поля кода, исходя из которого можно получить адреса всех других ее полей. Поскольку именно этот адрес компилируется в шитый код, то он также называется *адресом компиляции*.

Поле имени содержит имя слова в виде строки со счетчиком. Стандарт ограничивает максимальную длину имени 31 литерой (5 двоичными разрядами), чтобы остающиеся разряды в байте счетчика можно было использовать под специальные признаки. Важнейший из них — признак немедленного исполнения, который обычно занимает старший разряд байта-счетчика. Таким образом, поле имени имеет переменную длину, которая определяется значением счетчика.

Поле связи занимает 2 байта и содержит адрес заголовка предыдущей словарной статьи. Через это поле словарные статьи связаны в цепные списки, в которых можно вести поиск статьи по имени слова. Поле связи первой статьи в списке содержит некоторое специальное значение, обычно нуль. Новые слова добавляются в конец списка, и поиск слов ведется от конца списка к началу.

Тело словарной статьи реализуется через шитый код в одной из его разновидностей. Для определенности примем за основу косвенный шитый код. В этом случае интерпретатор шитого кода называется *адресным интерпретатором* форт-системы, поскольку интерпретирует последовательность адресов, каждый из которых является адресом компиляции некоторой словарной статьи (адресом ее поля кода). В качестве собственных данных адресный интерпретатор использует стек возвратов и указатель на текущее место в интерпретируемой последовательности адресов.

Поле кода словарной статьи в случае косвенного шитого кода занимает 2 байта и содержит адрес машинной программы, которая и выполняет действие, связанное с данным словом. Точка NEXT адресного интерпретатора обеспечивает этой программе доступ к полю параметров данной словарной статьи: Для статей, соответствующих определению через двоеточие, поле кода содержит адрес точки CALL адресного интерпретатора, а поле параметров представляет собой последовательность адресов словарных статей, входящих в данное определение. Завершается такая последовательность адресом словарной статьи EXIT (выход), который компилируется завершающей данное определение точкой с запятой. Для словарных статей нижнего уровня, т. е. реализованных непосредственно в машинном коде, поле кода содержит адрес соответствующей машинной программы, которая обычно располагается в поле параметров этой статьи. Таким образом, можно сказать, что поле кода словарной статьи содержит адрес машинной программы — интерпретатора поля параметров. Все слова, определенные через двоеточие, имеют в качестве интерпретатора действие CALL адресного интерпретатора, слова нижнего уровня наоборот имеют каждое свой отдельный интерпретатор в виде машинной программы, размещенной в поле параметров. В частности, такой программой для слова EXIT является действие RETURN адресного интерпретатора.

На рис. 2.4. приведены адресный интерпретатор и структура словарной статьи для определения

```
: F  ( A ---> A*[A+1]/2)    DUP  1+  2  */  ;
```

вычисляющего сумму натуральных чисел от I до А по известной формуле. В качестве языка нижнего уровня для записи действий адресного интерпретатора применяется очевидная нотация. Переменная RI обозначает указатель текущего места в интерпретируемом коде, процедура MEM дает значение, находящееся по заданному адресу в памяти форт-системы. Процедуры RPUSH и RPOP используются для обращения к стеку возвратов с тем, чтобы положить значение на его вершину и снять верхнее значение.

На рис. 2.4 изображены две словарные статьи: верхнего уровня для слова F, определенного через двоеточие, и нижнего уровня для слова EXIT, являю-

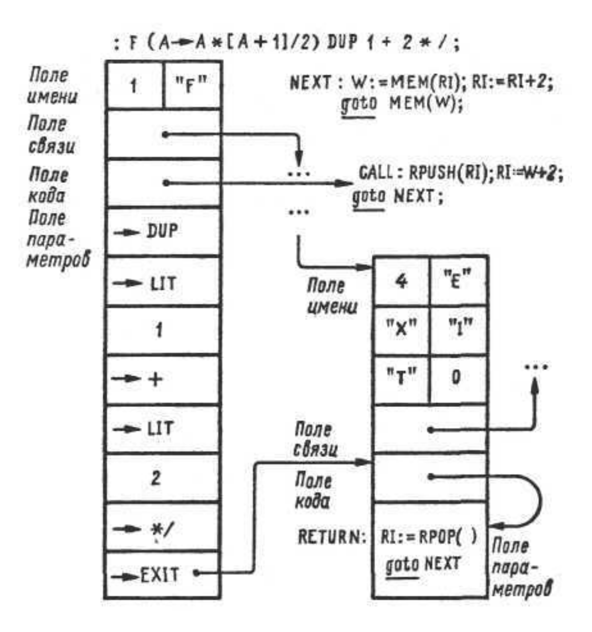


Рис. 2.4. Адресный интерпретатор и структура словарной статьи для косвенного шитого кода

щаяся к тому же частью адресного интерпретатора.

Поле имени каждой статьи начинается байтом-счетчиком, в котором записано число литер в имени слова. Далее располагаются коды самих этих литер.

Вслед за полем имени идет поле связи, содержащее адрес поля имени предыдущей словарной статьи. В тех реализациях, где двухбайтные значения должны располагаться с четного адреса, поле имени тоже располагают с четного адреса, дополняя его (как в случае слова EXIT) до четного числа байт специальным кодом (обычно пробелом или нулем).

Поле кода словарной статьи содержит адрес машинной программы. Для слова F это адрес точки CALL адресного интерпретатора, для слова EXIT — адрес его поля параметров, где и располагается соответствующая программа.

Наконец, в соответствии с техникой шитого кода в поле параметров статьи для слова F располагается последовательность адресов полей кода словарных статей тех слов, из которых составлено его определение.

В этих адресах перед именем слова стоит знак →. Скомпилированное число занимает две ячейки: в первой находится адрес специального служебного слова LIT (от LITERAL — литерал), а во второй — значение данного числа. Действие слова LIT состоит в том, что значение числа кладется на стек данных, а указатель интерпретации передвигается дальше, обходя это значение в шитом коде:

```
LIT: W:=MEM(RI); PUSH(W); RI:=RI+2; goto NEXT;
```

Здесь используется тот факт, что в момент перехода на очередную машинную программу в действие NEXT указатель текущего места RI уже установлен на следующий элемент интерпретируемой последовательности адресов.

Адреса полей словарной статьи имеют традиционные обозначения, которые представляют аббревиатуры их английских названий:

```
NFA  -  Name Field Address  -  адрес поля имени
LFA  -  Link Field Address  -  адрес поля связи
CFA  -  Code Field Address  -  адрес поля кода
PFA  -  Parameter Field Address  -  адрес поля
                                    параметров
```

Представлением словарной статьи считается адрес ее поля кода. Стандартное слово > BODY (от body - тело) CFA → PFA преобразует его в адрес поля параметров. По аналогии во многих реализациях введен следующий ряд слов для переходов между остальными полями статьи:

```
BODY>    PFA ---> CFA
>NAME    CFA ---> NFA
NAME>    NFA ---> CFA
N>LINK   NFA ---> LFA
L>NAME   LFA ---> NFA
```

Их реализация определяется принятой структурой словарной статьи.

Стандарт предусматривает слово EXECUTE (исполнить) CFA → которое снимает со стека адрес поля кода словарной статьи и исполняет ее. Непосредственно под этим значением в стеке должны находиться параметры, необходимые данному слову. Такой механизм открывает широкие возможности для передачи слов-команд в качестве параметров.

2.3. Стек возвратов и реализация структур управления

Один из важных принципов языка Форт состоит в том, чтобы предоставить программисту максимальный доступ ко всем средствам его реализации. В соответствии с этим принципом собственные данные адресного интерпретатора — стек возвратов — были сделаны доступными, для чего введены специальные слова: >R A →, R> →A и R@→A. Буква R (от RETURN — возврат) в имени этих слов напоминает о том, что они работают со стеком возвратов. Все эти слова можно использовать только внутри компилируемых определений. Во время исполнения любого такого определения, представленного в шитом коде как подпрограмма верхнего уровня, на вершине стека возвратов находится адрес того места, откуда данное определение было вызвано (адрес возврата). Это значение было помещено туда действием CALL при входе в данное определение. Адрес возврата будет снят со стека возвратов и использован для продолжения интерпретации действием RETURN, составляющим семантику слова EXIT.

Перечисленные слова позволяют программисту вмешиваться в описанный механизм вызова, реализуя свои собственные схемы передач управления, и, кроме того, использовать стек возвратов для хранения временных данных в пределах одного определения (все положенные на стек возвратов значения должны быть сняты с него перед выходом из определения). Однако неосторожное манипулирование стеком возврата может вызвать непредсказуемые последствия, вплоть до разрушения форт-системы. Слово >R (читается «на эр») переносит значения с вершины стека данных на стек возвратов. Обратное действие выполняет слово R> (читается «с эр»). Слово R@ (читается «эр разыменовать») копирует вершину стека возвратов на стек данных, сохраняя это значение на стеке возвратов.

С помощью описанных слов легко определить, например, приведенное выше слово LIT , компилируемое в шитый код вместе со следующим за ним значением, чтобы со время счета положить это значение на стек данных:

```
: LIT ( ---> A )  R@  @  R>  2+  >R ;
```

Во время работы данного определения слово R@ кладет на стек адрес возврата, указывающий в этот момент на следующий после адреса статьи LIT элемент в интерпретируемой последовательности адресов (см. рис. 2.4). Слово @ разыменовывает этот адрес, таким образом на стеке оказывается скомпилированное в шитый код число. Слова R> 2+ > R увеличивают адрес возврата на 2, чтобы обойти значение числа, т. е. чтобы оно не было воспринято как адрес некоторой статьи. Таким образом, в приведенном определении слова LIT реализован интересный способ передачи параметра через адрес возврата, невозможный в традиционных языках программирования.

Рассмотренные ранее структуры управления — условный оператор и циклы — также легко выражаются через эти понятия. По их образцу программист может создавать собственные структуры управления, совершенствуя свой инструментарий. Реализация условного оператора и циклов с проверкой опирается на следующие слова, аналогичные рассмотренному выше слову LIT:

```
: COMPILE  ( ---> )  R@  @  ,  R>  2+  >R  ;
: BRANCH   ( ---> )  R>  @  >R  ;
: ?BRANCH ( A--->) R> SWAP IF 2+ ELSE @ THEN >R ;
```

Слово COMPILE (компилировать) компилирует (т. е. добавляет) на вершину словаря значение, находящееся в шитом коде непосредственно за данной ссылкой на статью COMPILE. Слово BRANCH (переход) переустанавливает указатель интерпретации по адресу, скомпилированному вслед за данной ссылкой на статью BRANCH . Наконец, слово PBRANCH снимает значение со стека и анализирует его: если это ЛОЖЬ (нуль), то оно работает, как BRANCH ,— указатель интерпретации устанавливается по адресу, скомпилированному вслед за данной ссылкой на статью PBRANCH , а если это ИСТИНА (не нуль), то при интерпретации данной последовательности скомпилированный адрес перехода будет обойден. То обстоятельство, что в определении слова PBRANCH используется условный оператор, для реализации условного оператора несущественно, потому что на практике слова BRANCH и PBRANCH реализуются в форт-системах как подпрограммы нижнего уровня. Приведенные определения

лишь иллюстрируют выразительные возможности языка, показывая, что даже такие, самые элементарные действия можно задать машиннонезависимым способом в виде высокоуровневых определений.

Вот еще пример определения важного стандартного слова:

```
: LITERAL  ( A ---> A/  )   STATE @
     IF  COMPILE  LIT  ,  THEN  ;  IMMEDIATE
```

Слово LITERAL анализирует текущее состояние текстового интерпретатора: если это компиляция, то компилирует значение, находящееся на вершине стека, как литерал в шитый код; в противном случае это значение остается на стеке.

Теперь у нас достаточно средств, чтобы выразить стандартные структуры управления, как обычные определения, через двоеточие. Например, слова для условного оператора можно определить так:

```
: IF ( ---> A) COMPILE ?BRANCH  HERE  2 ALLOT ;
                                      IMMEDIATE
: THEN ( A ---> )  HERE  SWAP  !  ;  IMMEDIATE
: ELSE ( A1 ---> A2 ) COMPILE  BRANCH  HERE
     2 ALLOT  HERE  ROT  !  ;  IMMEDIATE
```

Все эти слова имеют признак немедленного исполнения, который придается им словом IMMEDIATE , исполненным сразу после завершения каждого определения. Это означает, что данные слова будут исполняться, а не компилироваться, как остальные, во время обработки тела определений текстовым интерпретатором форт-системы.

На рис. 2.5 показана последовательность состояний словаря и стека в разные моменты обработки фрагмента A B IF C D THEN E F в теле определения через двоеточие. Эту обработку выполняет текстовый интерпретатор, находящийся в состоянии компиляции. В результате строится последовательность адресов словарных статей в соответствии с принятой техникой шитого кода.

Пусть слова A, B, ..., F являются обычными форт-словами, не имеющими признака немедленного исполнения. Тогда соответствующие им адреса будут компилироваться на вершину словаря.

Состояние / на рис. 2.5. соответствует моменту обработки непосредственно перед вводом слова IF.

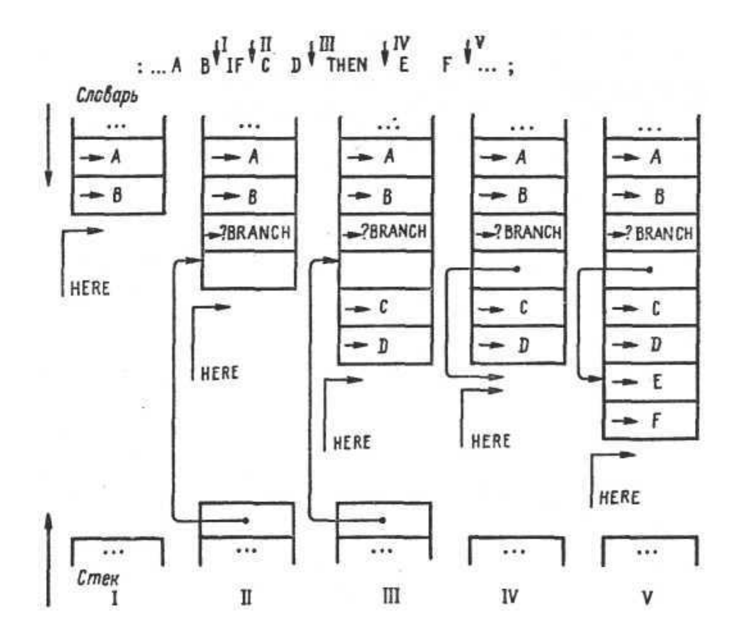


Рис. 2.5. Компиляция условного оператора

На вершине словаря скомпилированы адреса статей А и В и указатель вершины HERE указывает на следующий адрес. Слово IF имеет признак немедленного исполнения, поэтому будет не скомпилировано, а исполнено.

Состояние // показывает результат исполнения слова IF . На вершину словаря скомпилирована ссылка на статью ?BRANCH, вслед за которой отведено еще 2байта под адрес перехода, и адрес зарезервированного места сохранен на стеке данных.

Дальнейшие слова С и D будут опять компилироваться. Состояние /// соответствует моменту перед вводом слова THEN .

Слово THEN , как и IF, имеет признак немедленного исполнения, поэтому будет исполняться; в результате возникнет состояние *IV*. В этот момент определяется адрес перехода для скомпилированной словом IF ссылки на статью ?BRANCH ; это текущее значение указателя вершины словаря HERE , которое и вписывается в зарезервированные ранее 2 байта. Результат дальнейшей компиляции приводит к состоянию *V*.

Аналогичным образом исполняется и определение слова ELSE , которое компилирует обход части «иначе» и вписывает адрес ее начала в качестве адреса перехода для ссылки ?BRANCH . В свою очередь, адрес перехода для скомпилированного обхода будет вписан завершающим условный оператор словом THEN.

Можно повысить надежность программирования условного оператора введением контроля за соответствием слов IF , THEN и ELSE с помощью вспомогательного слова ?PAIRS :

```
: ?PAIRS ( A1,A2--->) - ABORT" НЕПАРНЫЕ СКОБКИ" ;
```

которое снимает два значения со стека и, если они не равны между собой (их разность не нуль), выдает сообщение об ошибке с пояснительным текстом «Непарные скобки». Стандартное слово ABORT" (выброс и кавычка) A → (исполнение) по своему употреблению аналогично слову ." (точка и кавычка): оно снимает значение со стека и, рассматривая его как логическое, сигнализирует об ошибке, печатая следующий за ним текст до кавычки, если это значение ИСТИНА (не нуль). Усиленные таким контролем определения слов условного оператора выглядят следующим образом:

```
: IF ( ---> A,1 )
     COMPILE ?BRANCH HERE 2 ALLOT 1 ; IMMEDIATE
: THEN ( A,1 ---> ) 1 ?PAIRS HERE SWAP ! ; IMMEDIATE
: ELSE ( A1,1 ---> A2,1 ) 1 ?PAIRS
     COMPILE BRANCH HERE 2 ALLOT
     HERE ROT ! ; IMMEDIATE
```

В этих определениях для контроля вместе с адресом зарезервированного места передается число 1, которое проверяется с помощью слова ?PAIRS в словах, использующих переданный адрес. Такой простой способ контроля на практике оказывается вполне достаточным. При этом программист может встроить любой другой контроль по своему желанию.

Приведенное определение условного оператора связано с реализацией стандартных слов BRANCH и ?BRANCH , выполняющих переходы в шитом коде. Из соображений эффективности эти слова обычно задают как подпрограммы нижнего уровня в машинном языке. Тогда в зависимости от архитектуры ЭВМ может оказаться предпочтительней ле абсолютный, как

в приведенной реализации, а относительный адрес перехода, компилируемый в шитый код сразу после ссылки на статью BRANCH или ?BRANCH . Чтобы сделать определения, использующие эти слова, машиннонезависимыми, стандарт предусматривает следующие слова для организации таких ссылок:

```
>MARK       ---> A
>RESOLVE    A --->
<MARK       ---> A
<RESOLVE    A --->
```

Слово > MARK (от MARK — отметить) резервирует место для ссылки вперед и оставляет адрес зарезервированного места на стеке. Слово > RESOLVE (от RESOLVE — разрешить) снимает этот адрес со стека и вписывает в него ссылку на текущую вершину словаря в соответствии с принятой реализацией переходов в шитом коде, согласованной с реализацией слов BRANCH и ?BRANCH . Аналогично слова <MARK и <RESOLVE предназначены для организации ссылок назад. Слово <MARK кладет на стек текущий адрес вершины словаря, а слово <RESOLVE компилирует ссылку на переданную точку. Окончательно определения слов условного оператора можно задать следующим образом (как они и выглядят в большинстве практических реализаций):

```
: IF ( ---> A,1 )  COMPILE  ?BRANCH  >MARK  1  ;
                                          IMMEDIATE
: THEN ( A,1--->) 1 ?PAIRS  >RESOLVE ; IMMEDIATE
: ELSE ( A1,1--->A2,1) 1 ?PAIRS  COMPILE  BRANCH
     >MARK  SWAP  >RESOLVE  1  ;  IMMEDIATE
```

Аналогичным образом реализуются слова для циклов с проверкой (рис. 2.6):

```
: BEGIN ( ---> A,2 )  <MARK  2  ;  IMMEDIATE
: UNTIL ( A1,2 ---> ) 2  ?PAIRS
     COMPILE  ?BRANCH  <RESOLVE  ;  IMMEDIATE
: WHILE ( A1,2 ---> A1,A2,3 ) 2  ?PAIRS
     COMPILE  ?BRANCH  >MARK  3  ;  IMMEDIATE
: REPEAT ( A1,A2,3--->) 3 ?PAIRS COMPILE BRANCH
     SWAP  <RESOLVE  >RESOLVE  ;  IMMEDIATE
```

Очевидно, что реализованные таким образом стандартные структуры управления могут произвольно глубоко

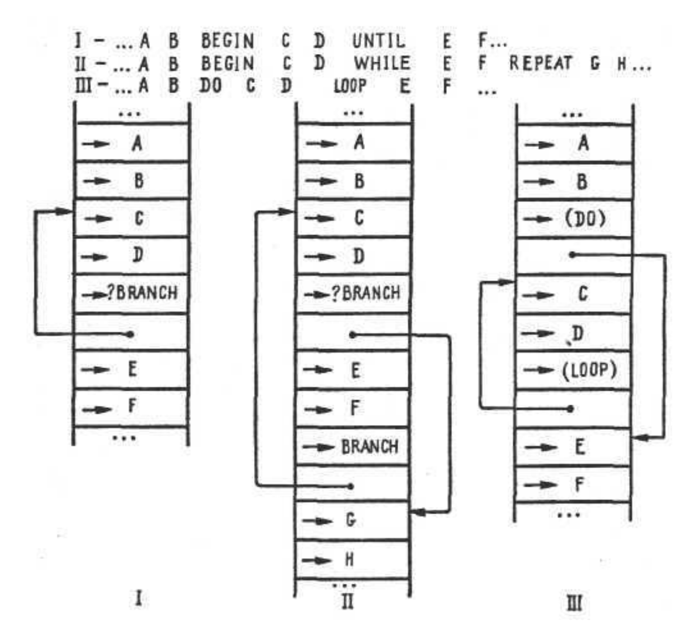


Рис. 2.6. Компиляция циклов

вкладываться друг в друга; несоответствие скобочных структур будет немедленно замечено, и программист получит сообщение об ошибке.

Циклы со счетчиком (рис. 2.6) реализуются аналогичным образом через вспомогательные слова (DO) , (LOOP) и (+ LOOP) , компилируемые в шитый код вместе с адресом перехода:

```
: DO ( ---> A1,A2,4 )  COMPILE  (DO)
     >MARK  <MARK  4  ;  IMMEDIATE
: LOOP ( A1,A2,4 --->) 4 ?PAIRS  COMPILE (LOOP)
     <RESOLVE  >RESOLVE  ;  IMMEDIATE
: +LOOP ( A1,A2,4--->) 4 ?PAIRS  COMPILE (+LOOP)
     <RESOLVE  >RESOLVE  ;  IMMEDIATE
```

Слово (DO) , с которого начинается исполнение скомпилированного цикла со счетчиком, переносит на стек возвратов следующий за ним адрес конца цикла (он нужен для немедленного выхода из цикла по слову LEAVE) и параметры данного цикла — начальное и граничное значения счетчика, снимая их с вершины

стека данных. Эти значения находятся на вершине стека возвратов в течение всего времени исполнения тела цикла.

Слово (LOOP) , завершающее скомпилированный цикл, продвигает текущее значение счетчика и в случае повторения цикла переводит указатель интерпретации на начало тела по скомпилированному вслед за ним адресу перехода, а при завершении цикла сбрасывает стек возвратов в исходное состояние.

Аналогично работает и слово (+LOOP) , которое дополнительно снимает со стека данных значение шага цикла. Разумеется, реализация этих слов должна соответствовать принятому способу задания переходов в шитом коде. Для прямых адресов перехода соответствующие определения можно задать так:

```
: (DO) ( A2:ГРАНИЧНОЕ,A1:НАЧАЛЬНОЕ ---> )
  R> ( A2,A1,R:ВОЗВРАТ)  DUP  @  >R ( ДЛЯ LEAVE)
  ROT  >R ( ГРАНИЧНОЕ)  SWAP  >R  ( НАЧАЛЬНОЕ)
  2+  >R   ( ОБОЙТИ АДРЕС В ШИТОМ КОДЕ)  ;
: (LOOP) ( ---> )
  R>  R>  R@ ( R:ВОЗВРАТ,I:ТЕКУЩЕЕ,A2:ГРАНИЧНОЕ)
  -  0  1.  D+  ( R,I-A2+1,F:ПРИЗНАК ЗАВЕРШЕНИЯ)
  IF  ( ЗАКОНЧИТЬ)
    DROP  R>  R>  2DROP  2+  ( ОБОЙТИ АДРЕС )
  ELSE ( ПРОДОЛЖИТЬ)
    R@  +  >R  ( НОВОЕ ЗНАЧЕНИЕ СЧЕТЧИКА)
    @  ( АДРЕС НАЧАЛА ТЕЛА ЦИКЛА)
  THEN  >R  ;
```

Определение (-(-LOOP) выглядит аналогично.

Во всех приведенных примерах доступ к адресу возврата в шитом коде осуществляется через стек возвратов из данного определения. Этот адрес модифицируется словами 2+ или @ , тем самым обеспечивая обход скомпилированной ссылки или переход по ее значению. Слова I и LEAVE , которые используются внутри тела цикла для получения текущего значения счетчика и немедленного выхода из цикла, можно задать так:

```
: I  ( ---> A )  R>  R@  SWAP  >R  ;
: LEAVE ( ---> )  R>  DROP  R>  DROP  R>  DROP  ;
```

Для повышения быстродействия практические реализации языка Форт обычно определяют все эти слова,

как и BRANCH , в виде подпрограмм нижнего уровня в машинном коде. В этом случае, например, слово I полностью эквивалентно слову R@ .

Описанная реализация циклов со счетчиком через использование стека возвратов накладывает ограничения при программировании. Неосмотрительное включение слов > R, R> , R@ и EXIT в тело цикла со счетчиком может привести к непредсказуемому результату.

Вместе с тем отстутствие какого-либо контроля является еще одной характерной чертой языка Форт. Благодаря этому программист может реализовать любые конструкции (включая и средства контроля).

2.4. Управление поиском слов

Множество слов, «известных» форт-системе, хранится в словаре в виде одного или более цепных списков словарных статей, соединенных через поле связи. Порядок поиска статей в этих списках обратен порядку их включения в словарь по времени определения: статья последнего определенного слова находится в начале списка. Такой порядок делает естественным исключение слов из словаря с помощью слова FORGET : нужный список просто усекается до соответствующего места.

Являясь самостоятельной структурой данных, список слов имеет и соответствующее определяющее слово — VOCABULARY (словарик, список слов) → аналогичное слову VARIABLE. Оно выделяет из входной строки очередное слово и определяет его как новый список слов, например, VOCABULARY A.

Со списками слов тесно связаны две стандартные переменные CONTEXT (контекст) и CURRENT (текущий) и слово FORTH (Форт), обозначающее список, который состоит из всех стандартных слов и включает, в частности, само это слово.

Поиск каждого введенного слова начинается в списке, на который указывает переменная CONTEXT , затем в случае неудачи просматривается список — текущее значение переменной CURRENT . Стандарт требует, чтобы последним просмотренным списком всегда был список FORTH . Исполнение слова, обозначающего список, делает его текущим значением переменной

CONTEXT , т. е. первым списком, который просматривается при поиске слов.

Стандартное слово DEFINITIONS (определения)

```
: DEFINITIONS ( ---> )  CONTEXT @  CURRENT !  ;
```

устанавливает переменную CURRENT по текущему значению переменной CONTEXT , т. е. соответствующий список становится вторым на очереди для просмотра и одновременно тем списком, куда добавляются новые словарные статьи. Первоначально обе эти переменные установлены на один и тот же список FORTH . К этому состоянию приводит исполнение текста FORTH DEFINITIONS . Разумеется, в этом случае поиск слов будет состоять в однократном просмотре списка слов FORTH.

Приведенное выше описание определяет порядок поиска лишь в общих чертах. Детали стандарт оставляет на усмотрение разработчиков реализации. Обычно используется схема, приведенная на рис. 2.7. Для каждого списка в поле параметров его статьи строится заголовок «фиктивной» статьи для невозмож-

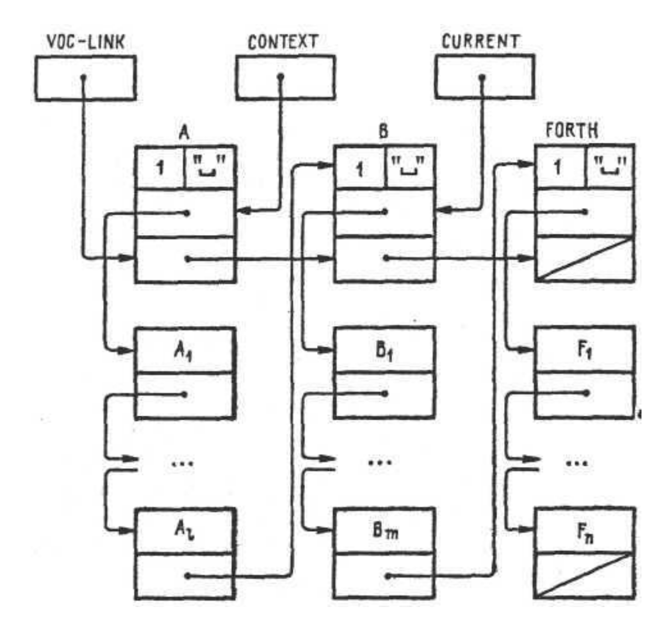


Рис. 2.7. Структура списков словарных статей

ного слова (например, состоящего из одного пробела), и представлением списка (значением переменных CONTEXT и CURRENT) служит адрес поля связи этого заголовка, т. е. адрес второго элемента в поле параметров. Сам этот элемент является входом в цепной список словарных статей, принадлежащих данному списку слов. В качестве его начального значения в момент создания списка словом VOCABULARY используется адрес поля параметров словарной статьи другого списка, который таким образом является базовым для данного (обычно это текущее значение переменной CONTEXT).

Таким образом, новый список через заголовок фиктивной статьи присоединяет к себе все слова базового списка.

В конечном счете любой список заканчивается словами списка FORTH , который уже не имеет базового. На рис. 2.7 показаны поля параметров списков A , B и FORTH . Список B является базовым для A и, в свою очередь, базируется на списке FORTH . Слова $A_1, \ldots,$ A_l входят в список A ; $B_1, \ldots, B_m$ — в B и $F_1, \ldots, F_n$ — в FORTH.

Показанный на рисунке порядок поиска слов соответствует исполнению текста B DEFINITIONS A, в результате чего сначала будут просмотрены списки A , B , FORTH , а затем B , FORTH .

Чтобы обеспечить большую гибкость в управлении порядком поиска слов и иметь возможность переопределять стандартные слова, в некоторых реализациях предусмотрено создание *безбазовых списков*. Это списки, создаваемые в контексте списка FORTH , для которых FORTH был бы базовым. Наличие безбазовых списков компенсируется тем, что в алгоритме поиска последним действием после просмотра списков, определяемых переменными CONTEXT и CURRENT, просматривается список FORTH , если он еще не был просмотрен через эти переменные.

Для учета всех существующих списков (что необходимо для реализации слова FORGET) в их поле параметров резервируется еще одна ячейка: через которую все эти структуры связаны в единый цепной список, начинающийся в служебной переменной VOC — LINK . Определение слова VOCABULARY с учетом перечисленных соглашений может выглядеть так:

```
: VOCABULARY  ( ---> )  CREATE
   256  BL  +  ,  CONTEXT  @  2-  ,
   HERE  VOC-LINK  @  ,  VOC-LINK  !
 DOES>  ( A ---> )  2+  CONTEXT  !  ;
```

Когда в список включается новая словарная статья, то в поле связи ее заголовка копируется значение из поля связи фиктивной статьи в поле параметров данного списка, а туда заносится адрес нового заголовка. Таким образом, можно определить слово LATEST (последний):

```
: LATEST  ( ---> NFA )  CURRENT  @  @  ;
```

которое возвращает адрес заголовка последней созданной словарной статьи (обычно это адрес поля имени). Через это слово становится очевидной, например, реализация слова IMMEDIATE:

```
: IMMEDIATE ( --->) LATEST C@ 128 OR LATEST C! ;
```

которое устанавливает в единицу признак немедленного исполнения в байте-счетчике последней созданной словарной статьи.

В соответствии с общими принципами языка Форт сам процесс поиска слова в словаре доступен программисту. Стандарт предусматривает для этого следующие слова:

```
                ---> CFA
['']            ---> CFA  (исполнение)
[COMPILE]       --->    (КОМПИЛЯЦИЯ)
FIND          A ---> CFA,N / A,0
```

Слово ' (апостроф, читается «штрих») вводит очередное слово и ищет его в словаре, возвращая адрес поля кода найденной статьи (если слово не найдено, то это считается ошибкой).

Слово ['] имеет признак немедленного исполнения и используется внутри определений через двоеточие, образуя вместе со следующим словом единую пару: во время исполнения адрес поля кода этого слова будет положен на стек данных.

Слово [COMPILE], уже встретившееся ранее, вводит и компилирует следующее слово независимо от признака немедленного исполнения.

Наконец, слово FIND (найти) позволяет формировать образец для поиска программным путем: его параметром является адрес строки со счетчиком, которая рассматривается как имя слова. В случае успеха FIND возвращает адрес поля кода его словарной статьи и значение N, характеризующее признак немедленного исполнения: 1, если признак установлен, и —1, если отсутствует. В случае неудачи слово FIND возвращает прежний адрес строки со счетчиком и значение 0 — сигнал о неудаче.

Разумеется, слова ' и ['] используют слово FIND:

```
: ' ( ---> A ) BL WORD FIND IF EXIT THEN
  COUNT TYPE -1 ABORT" ?" ;
: ['] ( ---> ) ' COMPILE LIT , ; IMMEDIATE
: [COMPILE] ( ---> ) ' , ; IMMEDIATE
```

которое, таким образом, является основным в определении порядка поиска слов.

В заключение рассмотрим определение стандартного слова FORGET , которое вводит очередное слово, ищет его в словаре и исключает из словаря вместе со всеми словами, определенными после него.

Служебная переменная FENCE (забор) защищает начальную часть словаря от случайного уничтожения. Она содержит адрес, отмечающий границу защищенной части (чтобы исключить слова из защищенной области нужно сначала понизить это значение):

```
: FORGET ( ---> )      >NAME ( NFA)
  DUP FENCE @ U< ABORT" ЗАЩИТА ПО FENCE"
  >R VOC-LINK @ ( N0:ВХОД В СПИСОК СПИСКОВ)
BEGIN R@ OVER U< WHILE ( N:ЗВЕНО СВЯЗИ СПИСКА)
     FORTH DEFINITIONS
     @ DUP VOC-LINK ! ( N1:ЗВЕНО СЛЕДУЮЩЕГО)
  REPEAT ( N1:ЗВЕНО ОСТАВШЕГОСЯ СПИСКА)
BEGIN DUP 4 - ( N:ЗВЕНО СВЯЗИ,L:ВХОД В СЛОВА)
     BEGIN N>LINK @ ( N,NFA:ОЧЕРЕДНОЕ СЛОВО)
  DUP R@ U< UNTIL ( N,NFA1:ОСТАВШЕЕСЯ СЛОВО)
OVER 2- ( N,NFA1,L:АДРЕС СВЯЗИ ФИКТИВНОЙ СТАТЬИ)
     ! @ ?DUP 0=
UNTIL R> ( NFA0) HERE - ALLOT ;
```

Сначала определяется адрес статьи исключаемого слова, проверяется, что она расположена выше границы защиты, и этот адрес сохраняется на стеке возвратов.

В ходе исполнения первого цикла перебираются все существующие статьи для списков слов и исключаются те, адреса которых оказались больше данного (в силу монотонного убывания адресов в цепном списке VOC — LINK цикл прекращается, как только адрес очередной статьи оказывается меньше данного). Во время исполнения второго цикла продолжается перебор оставшихся статей для списков слов, при этом в каждом списке отсекаются статьи, адреса которых больше данного (здесь опять используется монотонное убывание адресов словарных статей в пределах одного списка). В заключение указатель вершины словаря понижается до заданного адреса, тем самым освобождая память в словаре. Перед началом описанных действий переменные CONTEXT и CURRENT устанавливаются на список FORTH , чтобы их значение было осмыслено во все время дальнейшей работы.

2.5. Реализация определяющих слов

Механизм определяющих слов составляет одно из основных достоинств языка Форт, главную «находку» его создателей. С его помощью программист может вводить свои типы данных и структуры управления, задавая их внешнее синтаксическое оформление и внутреннюю семантическую реализацию. Исходный строительный материал программист может брать из сравнительно небольшого исходного запаса слов-команд языка или создавать сам.

Широкие выразительные возможности и вместе с тем компактность реализации этого механизма вытекают из того, что в его основе лежит фундаментальный принцип частичной параметризации. Его применению в традиционных языках программирования мешал громоздкий аппарат процедурного вызова, который считался в этих языках элементарным и неделимым действием. В языке Форт конструкция вызова разложена на отдельные составляющие и доступна «по частям», в частности имеется доступ к адресу возврата и всему динамическому контексту вызова.

В сочетании с единым механизмом передачи параметров через стек и компактной реализацией через шитый код это дает недостижимый для других языков уровень свертки понятий.

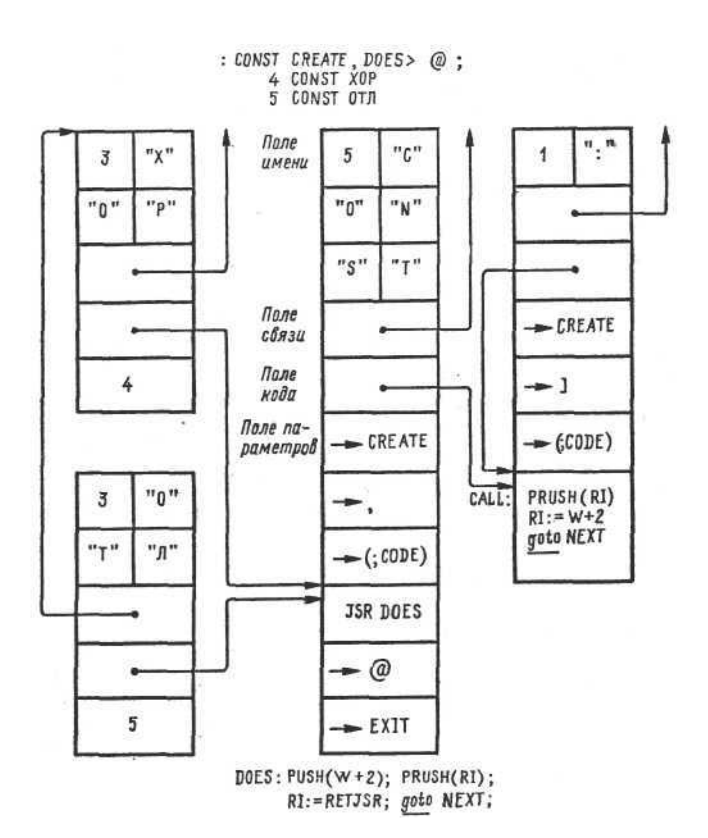


Рис. 2.8. Структура статьи определяющих и определяемых слов

На рис. 2.8 приведена структура словарных статей, возникающая после обработки определений

```
: CONST  CREATE  ,  DOES>  @  ;
     4  CONST  XOP      5  CONST  ОТЛ
```

Там же показана словарная статья для слова : , в теле которой находится точка CALL адресного интерпретатора.

Поле кода статьи CONST содержит адрес точки CALL, а поле параметров содержит последовательность ссылок на словарные статьи, скомпилированную адресным интерпретатором в соответствии с техникой шитого кода. Слово DOES> , входящее в это определение,

имеет признак немедленного исполнения, поэтому оно не компилируется вслед за ссылкой на статью для запятой, а исполняется. Его исполнение состоит в компиляции ссылки на статью вспомогательного слова (;CODE) и компиляции машинной команды JSR перехода с возвратом на специальную точку DOES в ядре форт-системы. Далее текстовый интерпретатор компилирует ссылку на статью для @ и исполняет (в силу признака немедленного исполнения) слово ;, которое завершает построение словарной статьи. Оно компилирует ссылку на статью EXIT и переключает текстовый интерпретатор в состояние исполнения.

Во время исполнения слова CONST во фрагменте 4 CONST XOP входящее в него слово CREATE создаст для слова XOP заголовок словарной статьи и поле кода, заслав туда стандартный адрес. Следующее слово , скомпилирует число 4 в поле параметров, а слово (;CODE) занесет в поле кода адрес следующей за ним машинной программы и закончит исполнение слова CONST :

```
: (;CODE)  ( ---> )  R>  LATEST  NAME>  !  ;
```

Значение, снимаемое с вершины стека возвратов словом R> ,— это как раз адрес команды JSR в определении слова CONST . Слово LATEST кладет на стек адрес заголовка последней созданной статьи, т. е. статьи XOP, а слово NAME> преобразует этот адрес в адрес поля кода. Поскольку со стека возвратов было снято одно значение, то по завершении данного определения управление вернется в точку после вызова определения, вызвавшего данное, т. е. на продолжение работы после вызова слова CONST. Аналогичные действия будут исполнены при обработке текста 5 CONST ОТЛ . Если теперь слово XOP будет исполняться, то из точки NEXT адресного интерпретатора управление будет передано на машинную программу по адресу из поля кода, т. е. на команду JSR в теле определения CONST . Эта команда перехода с возвратом передает управление на точку DOES, сообщив ей в качестве своего адреса возврата адрес следующей последовательности ссылок — исполняющей части определения CONST . Точка DOES кладет на стек адрес поля параметров статьи XOP (в этот момент в рабочей ячейке W еще находится адрес поля кода статьи XOP , загруженный туда действием NEXT) и исполняет действие CALL для исполняющей

части определения CONST . Следующее исполняемое слово @ заменит на стеке адрес поля параметров статьи XOP числом 4, скомпилированным по этому адресу, и затем слово EXIT завершит исполнение данного вызова слова XOP .

Слово : тоже является определяющим, и его словарная статья имеет такую же структуру. Рассмотрим его работу на примере трансляции определения CONST.

Создающая часть слова : состоит из слов CREATE и] . Первое выбирает из входной строки следующее за двоеточием слово (в данном случае CONST) и создает для него начало словарной статьи (заголовок и поле кода), а второе переключает текстовый интерпретатор в состояние компиляции. Последнее в создающей части слово (;CODE) вписывает в поле кода создаваемой новой статьи текущее значение указателя интерпретации, т. е. адрес точки CALL, которая располагается в теле данного определения, после чего исполнение двоеточия заканчивается. Поскольку теперь интерпретатор находится в состоянии компиляции, то следующие вводимые слова будут компилироваться, заполняя поле параметров статьи CONST последовательностью ссылок. Так будет продолжаться до тех пор, пока слово — точка с запятой, отмечающее конец определения и имеющее признак немедленного исполнения, не переключит интерпретатор обратно в состояние исполнения:

```
: ;  ( ---> )  COMPILE  EXIT  [COMPILE]  [  ;
                                     IMMEDIATE
```

Определяющие слова являются механизмом для создания классов слов со сходным «поведением», которое определяется исполняющей частью и включается в словарную статью слова через поле кода. Разница между словами внутри одного класса состоит в значении поля параметров, которое строится при определении слова. Слова, определенные через двоеточие, составляют один из классов наряду с константами, переменными, списками слов и другими, которые программист может вводить по своему усмотрению. При этом достигается существенная экономия памяти за счет того, что общая часть всех слов одного • класса присутствует в памяти в единственном экземпляре в статье определяющего слова.

Таким образом, в основе языка Форт лежат две структуры: *внешняя* — простейшая синтаксическая структура слова, представленного как последовательность литер, ограниченная пробелом, и *внутренняя* — структура словарной статьи, в которой поле кода задает интерпретатор поля параметров. Одним из частных случаев такого интерпретатора является адресный интерпретатор, и тогда поле параметров содержит последовательность интерпретируемых адресов в шитом коде. Этому случаю отвечает определение через двоеточие. Другие определяющие слова задают другие интерпретаторы и соответственно другие структуры поля параметров.

2.6 Встроенный ассемблер

Встроенный ассемблер позволяет программисту задавать реализацию слов непосредственно в машинном коде данной ЭВМ. Это дает ему возможность, с одной стороны, повышать быстродействие программы до максимально возможного уровня за счет перевода интенсивно используемых слов непосредственно в машинный код, а с другой — использовать особенности архитектуры данной ЭВМ и «напрямик» связываться с операционной системой и внешним окружением.

Примеры обоих случаев дает сама форт-система. Очевидно, что слова, выполняющие арифметические операции или действия с вершиной стека, естественно реализовывать непосредственно в машинном коде, поскольку скорость их исполнения в значительной степени определяет быстродействие системы. Вместе с тем слово 1+ , увеличивающее на единицу значение на вершине стека, можно определить как в машинном коде (особенно если в данной ЭВМ есть специальная команда увеличения значения на единицу), так и через двоеточие:

```
: 1+   ( A ---> A+1)  1  +  ;
```

В последнем случае, если к тому же слово 1 реализовано в виде отдельной словарной статьи 1 CONSTANT 1 , шитый код займет всего три ячейки, что может быть короче аналогичной программы в машинном коде. Разумеется, при этом исполнение будет дольше за счет интерпретации последовательности из трех ссылок вместо

прямого исполнения соответствующих машинных команд. Также очевидно, что, например, форт-слова для обмена с терминалом естественно задать на машинном уровне через обращения к соответствующим функциям операционной системы или непосредственно к аппаратуре.

Для определения слов непосредственно в машинном коде стандарт языка Форт предусматривает следующие слова:

```
ASSEMBLER  --->
CODE       --->
END-CODE   --->
;CODE      --->   (компиляция)
```

составляющие стандартное ассемблерное расширение обязательного набора слов.

Слово ASSEMBLER (ассемблер) является именем списка слов, содержащего слова, с помощью которых строится машинный код (мнемоники машинных команд, обозначения регистров, способы адресации и т. д.).

Слово CODE (код) является определяющим для слов нижнего уровня и обычно определяется так:

```
: CODE  ( ---> )
  CREATE  HERE  LATEST  NAME>  !  ASSEMBLER  ;
```

Оно используется в сочетании со словом END-CODE (конец кода): CODE <имя> <машинный-код> END-CODE, где «имя» является именем определяемого слова, а «машинный-код» — записью его реализации в машинном коде в соответствии с принятыми соглашениями.

Поле кода такой словарной статьи содержит адрес ее поля параметров, в котором располагается данный машинный код.

Наконец, слово ;CODE , имеющее признак немедленного исполнения, позволяет задавать исполняющую часть определяющих слов непосредственно в машинном коде:

```
;CODE  ( ---> )  COMPILE  (;CODE)
[COMPILE]  [  ASSEMBLER  ;  IMMEDIATE
```

Оно используется внутри определения через двоеточие для определяющего слова аналогично слову DOES>

```
: <имя> <создающая-часть>
          ;CODE <машинный-код> END-CODE
```

и отделяет высокоуровневую создающую часть от исполняющей части, заданной в машинном коде. Во время исполнения скомпилированного словом ;CODE слова (;CODE) адрес машинной программы, составляющей исполняющую часть, будет заслан в поле кода определяемого слова, которое таким образом получит интерпретатор, реализованный в машинном коде. На практике именно таким способом задают стандартные определяющие слова : , CONSTANT и VARIABLE .

Конкретный вид машинной программы зависит от архитектуры данной ЭВМ. Общим правилом является то, что этот текст представляет собой последовательность слов, которые исполняются текстовым интерпретатором, в результате чего на вершине словаря формируется соответствующий двоичный машинный код. Машинные команды записываются в естественной для Форта обратной польской форме: сначала операнды, а затем слово, обозначающее мнемонику команды.

Операнды — это слова, вычисляющие на стеке размещения операндов: номера регистров, адреса в памяти и их модификации, значения непосредственных операндов и т. д.

Мнемоника команды, обычно состоящая из традиционного обозначения данной команды и запятой, снимает со стека размещения своих операндов и компилирует соответствующий двоичный код.

Включение запятой в имя слова для кода команды, с одной стороны, подчеркивает тот факт, что данное слово компилирует команду, а с другой стороны, позволяет отличать, например, часто встречающиеся мнемоники ADD , CH и т. д. от чисел, заданных в шестнадцатиричной системе.

В табл. 2.1 приведена запись одних и тех же машинных команд в традиционном ассемблере и встроенном ассемблере разных форт-систем для нескольких распространенных типов ЭВМ. Как и в случае реализации структур управления, во встроенный ассемблер можно включить дополнительные средства контроля, которые, учитывая формат машинных команд, проверяют правильность задания их операндов.

Таблица 2.1. Сравнительная запись машинных команд в традиционном ассемблере и встроенном ассемблере форт-системы

Тип ЭВМ	Традиционный ассемблер	Встроенный ассемблер форт-системы
СМ-4	CMPB #12, (Rl) + JMP NEXT RTS	12 # Rl) + CMPB, NEXT JMP, RTS,
ЕС ЭВМ	STM 14,12,12(13) BALR 15, 0 B NEXT	14 12 12 (, 13 STM, 15 0 BALR, NEXT B,
K580	MOV A. B LXI H, 15 POP H	A B MOV, H 15 LXI, H POP,
БЭСМ-6	, UTC, =15 , XTA, 3, UTC, 777	0 0 5 # # UTC, 0 0 XTA, 3 777 UTC,

В гл. 3 рассмотрена реализация полного встроенного ассемблера для микропроцессора К580, занимающая 100 строк текста на языке Форт.

Встроенный ассемблер форт-систем часто делают «структурным», т. е. включают в него операторы ветвления и циклы, выполняющие переходы по значению управляющих разрядов в специальном регистре. По аналогии с такими же средствами языка Форт эти структуры задают с помощью тех же слов с добавлением, запятой: IF, THEN, ELSE, BEGIN, UNTIL и т. д. При этом вводят слова, обозначающие различные состояния управляющих сигналов, а слова, реализующие структурные операторы компилируют команды переходов, соответствующие указанным состояниям. Такой подход позволяет во многих случаях избежать введения сложного механизма переходов на метку, поскольку ассемблерные вставки, для трансляции которых и существует встроенный ассемблер, состоят, как правило, из нескольких команд.

Программа в машинном коде, заданная через слова CODE или ;CODE , получает управление от адресного интерпретатора и после ее завершения возвращает управление интерпретатору на точку NEXT. Для связи

с адресным интерпретатором в списке слов ASSEMBLER обычно предусматривается ряд констант, обозначающих точки входа, номера регистров и другие ресурсы адресного интерпретатора. Аналогичным образом предусматривается связь с операционной системой, встроенным постоянным программным обеспечением и аппаратурой.

В общем объеме больших программ на Форте ассемблерные коды составляют незначительную часть (в реализации самой форт-системы, например, 10—30 %), но вместе с тем такая возможность делает программиста максимально свободным в выборе средств для реализации своих идей.

2.7 Работа с внешней памятью

В настоящее время в каждом языке программирования тем или иным способом решаются вопросы обмена с внешней памятью. Из-за огромного разнообразия внешних устройств и способов хранения информации на внешних носителях единый универсальный механизм обмена отсутствует. В каждом языке определяется своя, в известной мере универсальная файловая система, которая при реализации моделируется на реальной файловой системе конкретной операционной системы.

Поскольку в язык Форт легко включать новые слова-команды, то надобность в стандартных универсальных развитых средствах обмена отпадает. В каждой реализации можно ввести такие средства исходя из особенностей и возможностей применяемых для данной ЭВМ файловой системы и конкретных внешних устройств. Вместе с тем желательно иметь механизм, обеспечивающий по крайней мере перенос и хранение на машинном носителе программных текстов на языке Форт. Исходя из этого стандарт предусматривает элементарнейшую файловую систему, которая легко реализуется в любой конкретной файловой системе или непосредственно через драйверы внешних устройств (магнитных дисков).

В языке Форт применяется механизм виртуальной внешней памяти, состоящей из блоков по 1 К байт каждый. Блок идентифицируется номером — числом в диапазоне от 1 до 32767. В адресном пространстве форт-

системы выделяется память под буферный пул, рассчитанный на одновременное хранение одного или более блоков. Каждый буфер помимо памяти для хранения собственно блока данных содержит номер блока и признак его измененности.

Слово BUFFER (буфер), получив номер блока, ищет свободный буфер в пуле и возвращает его адрес, записывая в служебную ячейку буфера переданный номер блока. Тем самым найденный буфер приписывается данному блоку, однако содержимое буферной памяти пока остается прежним. Если свободного буфера не оказалось, то аналогичным образом используется один из занятых. Если при этом в служебной ячейке буфера был установлен признак измененности, то данные из буферной памяти переписываются во внешнюю, они заменяют там прежнее содержимое соответствующего блока.

Слово BLOCK (блок) аналогично по использованию слову BUFFER за исключением того, что перепись данных из внешней памяти производится в приписанный данному блоку буфер. Разумеется, если данный блок уже находится в буферном пуле, то переписи данных не происходит, и слово BLOCK возвращает адрес этого буфера.

Наконец, слово UPDATE (изменить) устанавливает признак измененности последнего блока, к которому было адресовано обращение через слово BLOCK или BUFFER . Таким образом, впоследствии этот блок будет автоматически переписан во внешнюю память.

При реализации обмена с внешней памятью в качестве буферного пула обычно используется связный участок оперативный памяти. Пусть его начало задается константой FIRST , а конец — адрес байта, следующего за последним,— константой LIMIT. (Если пул располагается вплотную к концу адресного пространства, то этот следующий адрес равен нулю!) Буфера в пуле располагаются подряд, каждый начинается двухбайтной ячейкой, в которой записывается номер приписанного блока, причем старший разряд используется под признак измененности. Далее идет буферная память для блока размером 1024 байта, завершается буфер еще одной служебной ячейкой, в которой записан ноль (ее назначение указано в п.2.8). Пусть переменные PREV и USE указывают на текущий используемый

буфер и следующий, который будет выдан при запросе на свободный буфер. Определим слово +BUF, которое вычисляет адрес буфера, следующего в пуле за переданным, и возвращает признак несовпадения его с текущим:

```
: +BUF  ( A1 ---> A2,F )  1024  +  4  +
  DUP  LIMIT  -  IF  DROP  FIRST  THEN
  DUP  PREV  @  -  ;
```

Пусть служебные слова RBLK A,N → и WBLK A,N → выполняют чтение блока с указанным номером в заданную область оперативной памяти и запись из нее. Тогда с учетом принятых условий слова, выполняющие работу с внешней памятью, можно задать так:

```
: BUFFER  ( N ---> A  ПРИПИСАТЬ БЛОКУ N БУФЕР)
  USE  @  DUP  >R  ( ВЕРНЕМ ЭТОТ БУФЕР)
  BEGIN +BUF UNTIL  USE ! ( УСТАНАВЛ.СЛЕДУЮЩИЙ)
  R@  @  0<  ( ПРИЗНАК ИЗМЕНЕННОСТИ?)
  IF  R@  2+  R@  @  32767  AND  WBLK  THEN
  R@  !  ( ПРИПИСАЛИ НОВОМУ БЛОКУ)
  R@  PREV  !  R>  2+  ;
: BLOCK  ( N--->A:АДРЕС БУФЕРА С ДАННЫМИ БЛОКА)
  >R PREV @ DUP R@ - DUP + ( ТЕКУЩИЙ - ТОТ ЖЕ?)
  IF ( НЕТ)  BEGIN  +BUF  0=
   IF  DROP  R@  BUFFER DUP R@  RBLK  2- THEN
   DUP @  R@  -  DUP  +  0= UNTIL  DUP  PREV !
  THEN  R>  DROP  2+  ;
: UPDATE ( --->) PREV @ @  32768 OR  PREV @ ! ;
```

Для записи всех измененных буферов во внешнюю память служит слово SAVE-BUFFERS (сохранить буфера):

```
: SAVE-BUFFERS  ( ---> )
  LIMIT  FIRST  DO  I  @  32768  AND
     IF  I  @  32767  AND  DUP  I  !
     I  2+  SWAP  WBLK  THEN
  1028  +LOOP  ;
```

При исполнении слова SAVE-BUFFERS все буфера остаются приписанными прежним блокам. Слово FLUSH (очистить) переписывает все исправленные блоки во внешнюю память и освобождает буфера. Многие реализации имеют слово EMPTY-BUFFERS (опустошить

буфера), которое освобождает буферный пул, не переписывая исправленные блоки:

```
: EMPTY-BUFFERS  ( ---> )
  LIMIT  FIRST  DO  0  I  @  !  1028  +LOOP  ;
: FLUSH  ( ---> )  SAVE-BUFFERS  EMPTY-BUFFERS  ;
```

Внешняя память форт-системы в основном используется для хранения форт-текстов. Блок внешней памяти с форт-текстом называется *экраном* и условно разбивается на 16 строк по 64 литеры в каждой. Такой формат экрана сложился традиционно и закреплен в стандарте.

Программист создает и исправляет экраны во внешней памяти, используя встроенный редактор форт-системы. Как и в случае встроенного ассемблера, стандарт не определяет конкретный вид и состав его команд, поскольку это в значительной степени определяется функциональными возможностями и характеристиками конкретных терминалов и средствами работы с ними. Обычно слова-команды текстового редактора составляют отдельный список слов с именем EDITOR (редактор), базирующийся на списке FORTH. Для распечатки редактируемого экрана на терминале чаще всего используют слово LIST (распечатать), которое, кроме того, сохраняет номер экрана в служебной переменной SCR:

```
: LIST  ( N ---> )  DUP  SCR  !
  CR  ." ЭКРАН "  DUP  .  BLOCK
  16  0  DO  DUP  I  64  *  +
  CR  I  3  .R  SPACE  64  TYPE  LOOP  DROP  ;
```

Вначале печатается номер данного экрана, затем его строки. Перед каждой строкой печатается ее номер — число в диапазоне от 0 до 15. По завершении редактирования исправленные экраны можно записать во внешнюю память словом FLUSH .

2.8. Интерпретация входного потока

Собственно работа форт-системы заключается в распознавании и исполнении слов-команд, которые программист вводит с терминала. Ввод осуществляется построчно: набрав нужный текст, программист нажимает специальную управляющую клавишу, сообщая тем

самым о завершении ввода. Форт-система размещает введенный текст в специальном буфере для ввода с терминала, который располагается в адресном пространстве оперативной памяти. Адрес начала этого буфера дает стандартное слово TIB (сокращение от TEXT INPUT BUFFER — буфер для ввода текста), его длина хранится в стандартной переменной #TIB . Данный буфер представляет собой входной поток, из которого слово WORD выбирает слова. По исчерпании входного потока форт-система вводит в этот буфер новый текст, получаемый от программиста. При возникновении какой-либо ошибочной ситуации форт-система прекращает дальнейшую интерпретацию текущего содержимого этого буфера и, выдав соответствующее сообщение программисту, заполняет буфер новым текстом, который после этого вводит программист.

Альтернативой вводу текста с терминала является ввод из внешней памяти форт-системы. Переключением входного потока на внешнюю память и обратно на терминал управляет стандартная переменная BLK (сокращение от BLOCK — блок), значение которой проверяется каждый раз в слове WORD . Если это нуль, то в качестве входного потока служит буфер TIB , в противном случае это значение рассматривается как номер блока внешней памяти, который используется как входной поток (этот блок переносится в оперативную память словом BLOCK). Текущая позиция во входном потоке хранится в стандартной переменной >IN (от IN — вход) и в случае ввода с терминала изменяется в пределах от 0 до значения #TIB , а при вводе из внешней памяти — в диапазоне от 0 до 1024.

Обычно конец входного потока в оперативной памяти (отмечается нулевым кодом (именно для этого в буферном пуле после памяти для данных блока резервируется еще одна ячейка). Слово WORD , «натыкаясь» на нулевой код, возвращает в качестве результата пустую строку с нулевым значением счетчика литер, при этом в список FORTH включается словарная статья для такого «пустого» слова. Оно имеет признак немедленного исполнения и поэтому всегда исполняется назависимо от текущего состояния текстового интерпретатора. Его исполнение состоит в прекращении интерпретации данного входного потока и тем самым позволяет избежать дополнительных проверок на ис-

черпание. Обычно в реализациях языка Форт определяется слово INTERPRET (интерпретировать), выполняющее интерпретацию текущего входного потока. Оно представляет собой бесконечный цикл по вводу и исполнению (или компиляции) слов:

```
INTERPRET  ( ---> )  BEGIN  BL  WORD  FIND
?DUP  IF  ( ПРОВЕРИТЬ ПРИЗНАК IMMEDIATE)
1+  IF  EXECUTE
    ELSE  STATE  @  IF  ,  ELSE  EXECUTE  THEN
    THEN
ELSE  ( МОЖЕТ БЫТЬ ЭТО ЧИСЛО?)
    NUMBER  DPL  @  1+
    IF  [COMPILE]  2LITERAL
    ELSE  DROP  [COMPILE]  LITERAL  THEN
THEN  AGAIN  ;
```

В приведенном примере конструкция BEGIN-AGAIN определяет бесконечный цикл.

Слово AGAIN выполняет безусловный переход на начало цикла.

В случае если очередное введенное слово не найдено в словаре, исполняется слово NUMBER, которое пытается воспринять его как запись числа в соответствии с текущим основанием системы счисления — значением переменной BASE . Если это удалось, то слово NUMBER возвращает значение числа как значение двойной длины и дополнительно в переменной DPL сообщает позицию десятичной точки в нем (- 1 , если точки в записи числа не было). В противном случае возникает ошибочная ситуация «Слово не найдено», и интерпретация прекращается. Если же введенное слово оказалось числом, то в зависимости от наличия в нем точки оно рассматривается как число двойной или одинарной точности. Таким образом, пустое слово — ограничитель входного потока — прекращает исполнение слова INTERPRET и возобновляет исполнение слова, его вызвавшего:

```
: X  ( ---> )  R>  DROP  ;  IMMEDIATE
```

Здесь X обозначает пустое слово.

Таким образом, работу форт-системы можно задать таким бесконечным циклом:

```
: ФОРТ-СИСТЕМА  ( ---> )
  BEGIN  CR  ." >"  ( ПРИГЛАШЕНИЕ К ВВОДУ)
    TIB  80 EXPECT ( ВВЕСТИ ТЕКСТ С ТЕРМИНАЛА)
    SPAN  @  #TIB  ! ( УСТАНОВИТЬ ЕГО ДЛИНУ)
  0 TIB  #TIB @ +  C! ( УСТАНОВИТЬ ОГРАНИЧИТЕЛЬ)
  0  >IN !  0  BLK ! ( УСТАНОВИТЬ ВХОДНОЙ ПОТОК)
   INTERPRET ( ИНТЕРПРЕТИРОВАТЬ ВВЕДЕННЫЙ ТЕКСТ)
    STATE @  0=  IF ." OK" THEN ( ПОДТВЕРЖДЕНИЕ)
  AGAIN ;
```

Для переключения входного потока на внешнюю память стандарт предусматривает слово LOAD (загрузить) :

```
: LOAD ( N:НОМЕР--->  ИНТЕРПРЕТИРОВАТЬ ЭКРАН)
  >IN  @  >R  BLK  @  >R  ( СОХРАНИТЬ ТЕКУЩИЙ)
  BLK  !  0  >IN  !  ( УСТАНОВИТЬ НОВЫЙ)
  INTERPRET  ( ПРОИНТЕРПРЕТИРОВАТЬ ЕГО)
  R> BLK !  R> >IN ! ; ( ВЕРНУТЬСЯ К ПРЕЖНЕМУ)
```

Параметром слова LOAD является номер экрана (блока) для интерпретации. Очевидно, что приведенное нами выше определение допускает рекурсивное использование слова LOAD внутри экранов во внешней памяти.

Некоторые реализации предусматривают слово THRU (сквозь) для последовательной интерпретации диапазона экранов:

```
: THRU  ( N1,N2 --->  ИНТЕРПРЕТИРОВАТЬ ЭКРАНЫ)
        (           ОТ N1 ДО N2 ВКЛЮЧИТЕЛЬНО)
  1+  SWAP  DO  I  LOAD  LOOP  ;
```

Для продолжения интерпретации следующего экрана часто используется слово →, логически сцепляющее следующей экран с данным:

```
: -->  ( ---> )  BLK  @  0=
  ABORT" НЕДОПУСТИМОЕ ИСПОЛЬЗОВАНИЕ -->"
  0  >IN  !  BLK  @  1+  BLK  !  ;  IMMEDIATE
```

В этом случае интерпретации сцепленных экранов нужно «загрузить» словом LOAD только первый из этих экранов.

2.9. Целевая компиляция и модель форт-системы

Решая задачу на языке Форт, программист расширяет исходный набор слов-команд форт-системы, добавляя к нему новые определения. Его конечной целью является определение некоторого «главного» слова, исполнение которого и решает поставленную задачу. Главное слово можно запустить на счет через текстовый интерпретатор форт-системы, если ввести имя слова в качестве очередной форт-команды. Во время исполнения этого слова используются ранее введенные определения вспомогательных слов и (прямо или косвенно) ряд слов исходной форт-системы.

Во многих форт-системах предусмотрена возможность сохранения текущего состояния словаря во внешней памяти в качестве двоичного форт-модуля. Это позволяет при последующих обращениях к данной задаче сразу начинать счет, запуская на исполнение ее главное слово. При таком подходе естественно возникает задача минимизации используемых средств и построения конечного программного продукта, уже не зависящего от форт-системы. Такую задачу позволяет решать *пакет целевой компиляции,* который является ценным инструментальным средством при создании прикладного программного обеспечения на языке Форт.

Целевая компиляция позволяет создавать единый независимый рабочий модуль исходя из комплекса взаимосвязанных определений на языке Форт. Составной частью этого комплекса является запись определений всех стандартных слов, которая образует модель форт-системы и над которой «надстраиваются» определения данной задачи. В тексте модели обычно выделяется *запускающая часть,* которая обеспечивает инициирование работы и переход к исполнению главного слова в данном комплексе определений. Если в реализации есть пакет целевой компиляции, его часто используют для раскрутки самой форт-системы исходя из некоторой ее начальной версии. В этом случае главным является приводившееся выше слово ФОРТ-СИСТЕМА, которое выполняет бесконечный цикл по вводу входного потока с терминала и его текстовой интерпретации.

Входной текст для целевой компиляции помимо собственно текста на языке Форт содержит еще ряд слов-директив, позволяющих выполнять компиляцию за один проход по тексту. Эти директивы включают, в частности, средства предописания слов для того, чтобы некоторые слова можно было использовать раньше их определения (например, чтобы в запускающей части, с которой обычно начинается входной текст, указать имя главного слова, которое будет определено только в конце). Важной возможностью является управление построением целевой словарной статьи: некоторые или даже все статьи можно создать без заголовка, значительно сократив за счет этого размер рабочего модуля. Кроме того, язык директив обычно содержит средства для диагностики, распечатки карты создаваемого модуля, построения таблицы перекрестных ссылок, сбора и распечатки статистической информации об использовании тех или иных слов.

Пакет целевой компиляции представляет собой отдельно загружаемый пакет, написанный на языке Форт. В начале работы он резервирует область памяти для целевого адресного пространства, в которой строит двоичный код конечного программного продукта по мере обработки входного текста. После завершения целевой компиляции построенный модуль может быть выгружен во внешнюю память для дальнейшего использования в качестве независимого модуля в соответствии с заложенными в него соглашениями о связях.

В состав пакета целевой компиляции входят целевой ассемблер и целевой компилятор. *Целевой ассемблер* отличается от обычного встроенного ассемблера форт-системы только тем, что строит машинный код не в инструментальном адресном пространстве от текущей вершины словаря, которая дается словом HERE (здесь), а в целевом. Указатель в целевом адресном пространстве обычно обозначается словом THERE (там) внутри самого пакета, который пользуется обоими этими указателями. Для того чтобы получить целевой ассемблер из исходного инструментального, как правило, достаточно оттранслировать его в контексте других определений компилирующих слов (, , C, , ALLOT и т. д.), которые выполняют те же действия, но в целевом адресном пространстве, а не в инструментальном.

Задача *целевого компилятора* состоит в построении высокоуровневого шитого кода в целевом адресном пространстве, причем все ссылки на статьи, включая и ссылки на все стандартные форт-слова, должны также относиться к этому пространству. Для решения этой задачи обычно поступают следующим образом. В списке слов целевой компиляции переопределяют все стандартные определяющие слова (:, CONSTANT, VARIABLE, CODE и т. д.) таким образом, чтобы они строили две статьи: одну стандартным образом, но в целевом адресном пространстве, а другую — специальным образом — в инструментальном. Исполнение инструментальной статьи компилирует ссылку, соответствующую целевой статье данного слова, как очередной элемент шитого кода на вершину целевого словаря. Обрабатывая стандартным текстовым интерпретатором высокоуровневое определение в контексте таких инструментальных определений, мы получаем в целевом пространстве соответствующий шитый код. Аналогичным образом должны быть переопределены и все слова с признаком немедленного исполнения, реализующие структуры управления и выполняющие какую-либо компиляцию во время обработки определений (IF, THEN ,; и т. д.). Эти слова в отличие от их стандартных прототипов должны компилировать статьи и переходы для целевого адресного пространства, а не инструментального. Совокупность перечисленных определений и образует целевой компилятор.

При построении конечного программного продукта путем целевой компиляции возможно существенное уменьшение его размера. Кроме уже упоминавшегося исключения из словарных статей ненужных заголовков целевая компиляция позволяет вести учет всех используемых данной задачей слов и автоматически собрать для нее специальное ядро поддержки, состоящее только из тех слов, которые в ней действительно используются. При этом возможна еще большая оптимизация за счет непосредственной подстановки определений в место их вызова для тех слов, которые оказались использованными только один раз. Некоторые пакеты целевой компиляции дают возможность довести такую подстановку до уровня машинного кода, при этом высокоуровневые определения рассматриваются как своеобразные макроопределения. После

выполнения такой макрогенерации результирующий машинный код уже не требует адресного интерпретатора и представляет собой линейную последовательность непосредственно исполняемых машинных команд.

Именно этот прием был применен, например, при создании известного пакета машинной графики ГРАФОРТ для персонального компьютера фирмы ИБМ и компьютеров фирмы «Эппл» *(Apple,* США). Сравнительно большой объем получившегося машинного кода компенсировался чрезвычайно высоким быстродействием, недостижимым для традиционных языков программирования. В то же время совершенно очевидно, что разработать и отладить программу такого объема на языке ассемблера вручную практически невозможно.

Описанный подход к целевой компиляции применим и в том случае, когда целевая ЭВМ, для которой строится конечный программный продукт, отличается по своей архитектуре и системе команд от инструментальной. В этом случае изменения касаются только целевого ассемблера и определений в машинном коде, которые должны отвечать системе команд целевой ЭВМ. Такое использование целевой компиляции в качестве инструментального кросс-средства все чаще имеет место при создании программного обеспечения встроенных микропроцессоров и специализированных микропроцессорных устройств на их основе.

Глава 3

Примеры программных разработок

В этой главе приведены примеры реализации отдельных программных средств — небольших удобных расширений (инфиксная форма записи, локальные переменные, конструкция «переключатель», векторное поле кода) — и целых инструментальных систем на базе языка Форт. Основное внимание уделено методологическим аспектам использования Форта как расширяемого языка. Все приведенные примеры отражают опыт использования Форта в конкретных разработках.

3.1. Средства отладки форт-программ

Программы на языке Форт — определения слов — кодируются и вводятся в ЭВМ методом «снизу вверх», т. е. начиная с элементарных, использующих только стандартные слова форт-системы, и кончая определением главного слова, использующего уже введенные. Такой порядок естественным образом предполагает немедленную отладку вводимых определений, поскольку определение готово к исполнению сразу же после ввода. Отладка облегчается тем, что механизм взаимодействия модулей упрощен до предела: через стек данных, который программист может сам заполнить значениями параметров перед вызовом отлаживаемого слова.

Для распечатки текущего состояния стека данных и стека возвратов многие реализации имеют слова S. и R. . В сочетании со словом DUMP , которое распечатывает область памяти, и словом ?

```
: ? ( A ---> ) @ . ;
```

которое печатает значение, находящееся по указанному адресу, эти слова создают богатые возможности для диалоговой отладки форт-слов.

То обстоятельство, что в основе языка Форт лежит интерпретатор и при исполнении каждого слова управление проходит через точку NEXT адресного интерпретатора форт-системы, дает дополнительные возможности для организации автоматического слежения за исполнением скомпилированных слов. Программист может организовать *динамическую подмену точки NEXT* на такую, которая, выполняя те же действия по переходу к интерпретации очередной ссылки, выполняет и заданные отладочные действия. Поскольку в момент исполнения точки NEXT через адрес поля кода имеется доступ к адресу поля имени словарной статьи, то отладочные действия можно формулировать, задавая имена интересующих программиста слов.

Например, по специальному слову ON-NEXT (ПРИ-СЛЕД) строится список слов (в виде списка адресов компиляции), подлежащих отслеживанию. В частности, его можно задать, как все слова, определения которых расположены в словаре выше некоторого адреса. Для всех этих слов указываются отладочные действия, состоящие, как правило, из распечатки имени слова и стека данных в момент перехода к его исполнению. В некоторых случаях можно проводить анализ стека возвратов и отслеживать возврат из определений, распечатывая результаты из стека данных на момент возврата. Специальные слова TRACE-ON (СЛЕЖ-ВКЛ) и TRACE-OFF (СЛЕЖ-ВЫКЛ) включает и выключают механизм отладочного слежения, подменяя или восстанавливая точку NEXT адресного интерпретатора.

С помощью этого же механизма можно ввести защиту от зацикливания, например подсчитывая в точке NEXT число исполненных слов и возобновляя диалог при достижении некоторого заданного значения этого счетчика. Если данная ЭВМ имеет встроенный таймер и операционная система позволяет обрабатывать асинхронные выходы по истечении заданного интервала времени, то этот механизм можно также использовать для предохранения от зацикливания. В процедуре обработки выхода по исчерпанию временного интервала нужно подменить точку NEXT на такую, которая выполнит возврат к диалогу с программистом. При этом программист может получить исчерпывающую информацию о месте такого «прерывания от таймера».

Другой полезный механизм отладочного слежения позволяет переключаться на диалог с программистом в момент исполнения заданного слова и реализуется через *подмену поля кода в заданной словарной статье.* Для выполнения такой подмены нужно определить специальное слово, например, STOP (стойте!), которое выбирает имя слова из входного потока и засылает в поле кода его словарной статьи адрес специального кода. Прежнее значение поля кода сохраняется в специально резервируемой для этого ячейке словаря. Новый код для подмененного таким образом слова состоит в переключении на текстовую интерпретацию входного потока, так что все необходимые отладочные распечатки и установки значений программист задает непосредственно в диалоге. Слово GO (идите!) завершает диалог и продолжает исполнение программы, при этом отлаживаемое слово, вызвавшее останов, исполняется в прежнем виде (для этого используется сохраненное «настоящее» значение из его поля кода). Можно предусмотреть специальное слово для продолжения работы без исполнения рассматриваемого слова.

Описанные механизмы, являясь достаточно простыми, вместе с тем существенно облегчают отладку сложных программ. Их сильная зависимость от реализации не позволяет сделать эти определения переносимыми. Вместе с тем именно эта черта позволяет реализовать такие отладочные средства максимально удобным для программиста способом с учетом конкретных особенностей данной реализации и ее операционного окружения.

Удобным вспомогательным средством, позволяющим быстро находить место останова в терминах входного текста на языке Форт, является *символьная распечатка шитого кода.* Поскольку скомпилированная ссылка представляет собой адрес поля кода словарной статьи, то по ней через слово > NAME можно вычислить адрес поля имени и напечатать это имя словом ID. :

```
: ID. ( NFA --->) COUNT 31 AND TYPE SPACE ;
```

Слово COUNT в качестве счетчика длины выдает значение первого байта поля имени; поскольку старшие разряды этого байта используются под специальные признаки, то необходимо специальное преобразование, определяемое представлением счетчика, чтобы получить

истинную длину поля имени. Константа 31 как ограничение на длину имени слова дается стандартом языка.

В программе распечатки последовательности ссылок надо предусмотреть специальную обработку некоторых адресов, вслед за которыми скомпилирована не ссылка на очередную статью, а некоторое другое значение. Такими «особыми» ссылками в стандарте языка являются слова для выполнения переходов BRANCH и ?BRANCH, для реализации циклов (DO) , (LOOP) и (+LOOP) , для исполнения литералов LIT и 2LIT и некоторые другие.

3.2. Инфиксная запись формул

Для многих программистов трудным барьером на пути к овладению языком Форт оказывается используемая в нем обратная польская форма для записи выражений. Опишем простую надстройку над языком Форт, которая позволяет записывать формулы в обычной инфиксной форме с использованием скобок. Будем по-прежнему считать все элементы такого выражения (скобки, знаки операций и элементарные термы) отдельными форт-словами и разделять их при записи пробелами. Задача состоит в том, чтобы вычисления на стеке автоматически перегруппировывались исходя из инфиксной формы записи. Например, чтобы вместо текста 2 5 + 7 3 - * можно было писать ((2 + 5) * (7 + + 3)) . Внешние скобки нужны для того, чтобы отмечать конец выражения. При желании это можно задавать и каким-нибудь другим способом.

Для операций в инфиксной записи вводится понятие *приоритета* (старшинства), которое определяет порядок вычислений при отсутствии скобок. Приоритет обозначается целым числом, и операции с меньшим приоритетом выполняются после операций с большим приоритетом. Например, в выражении A+B/C подразумевается следующая расстановка скобок: (A+(B/C)), т. е. сначала выполняется деление и только потом сложение, потому что приоритет деления выше приоритета сложения. В случае равных приоритетов, например в выражении A + B + C, будем выполнять операции слева направо: ((A+B)+C). Традиционно используемые приоритеты двухместных операций показаны в табл. 3.1. Все одноместные операции (ABS, NEGATE,

Таблица 3.1. **Приоритеты двухместных операций**

Приоритет	2	3	4	5	6	7	8
Операция	OR XOR	AND	=	< >	+	/ MOD	**

NOT и др.) имеют максимальный приоритет (обычно 9).

Опишем вспомогательную структуру данных — стек ОПРЦ , элементами которого являются пары значений: приоритет операции и адрес кода, который ее вычисляет. Размер стека определяется максимальной глубиной вложенности формул, которую мы допускаем: CREATE ОПРЦ HERE 2+ , 40 ALLOT. Первым элементом в поле параметров слова ОПРЦ является указатель вершины этого стека (адрес первого свободного байта), далее зарезервирована память на 5 элементов по 4 байта (два значения) каждый. По аналогии со словами > R , R@ и R> определим слова для работы со стеком ОПРЦ:

```
: >ОПРЦ  ( A --->)  ОПРЦ  @  !  2  ОПРЦ +! ;
: ОПРЦ@  ( ---> A)  ОПРЦ  @  2-  @  ;
: ОПРЦ>  ( ---> A)  ОПРЦ@  -2  ОПРЦ  +!  ;
```

Обработка операций в инфиксной форме состоит в том, что операция не выполняется, а помещается вместе со своим приоритетом на стек операций, выталкивая из него на исполнение все операции с меньшим или равным приоритетом. В состоянии исполнения вытолкнутая операция исполняется, а в состоянии компиляции — компилируется. Предполагая, что в стеке ОПРЦ сначала размещается адрес поля кода для исполнения операции, а за ним приоритет, определим слово для такого выталкивания операций:

```
: >ОПРЦ> ( N:ПРИОРИТЕТ---> )  >R   BEGIN ОПРЦ@
    R@  <  NOT  WHILE  ОПРЦ>  DROP  ОПРЦ> ( CFA)
 STATE @ IF , ELSE EXECUTE THEN REPEAT R> DROP ;
```

Теперь введем определяющие слова для двухместных и одноместных операций и переопределим через них стандартные арифметические форт-слова:

```
: 2-ОП ( N:ПРИОРИТЕТ->)   >IN @ >R ' ( N,CFA)
    R> >IN !  CREATE  IMMEDIATE ( N,CFA) , ,
    DOES>  ( PFA->) 2@  ( N,CFA)
           >R >R R@ >ОПРЦ> R> R> >ОПРЦ >ОПРЦ ;
: 1-ОП ( ->)  9 2-ОП ;
2 2-ОП OR    2 2-ОП XOR    3 2-ОП AND    4 2-ОП =
5 2-ОП <     5 2-ОП >      6 2-ОП +      6 2-ОП -
7 2-ОП *     7 2-ОП /      7 2-ОП MOD
1-ОП NOT   1-ОП ABS   1-ОП NEGATE
```

В определении слова 2-ОП используется то обстоятельство, что код для исполнения данной операции обозначается словом, которое этим определяющим словом переопределяется. Поэтому перед тем как его имя будет выбрано из входного потока словом CREATE , текущая позиция во входном потоке запоминается на стеке возвратов и после исполнения слова ' (штрих) вновь устанавливается в то же положение для исполнения слова CREATE.

Переопределенные слова получают признак немедленного исполнения, чтобы описанные действия по переупорядочиванию вычислений выполнялись и во время компиляции, компилируя нужную последовательность операций. В заключение осталось переопределить круглые скобки, явно задающие порядок вычислений:

```
: (  0 >ОПРЦ ; IMMEDIATE
: )  1 >ОПРЦ> ОПРЦ> DROP ; IMMEDIATE
```

Открывающая скобка кладет на стек значение 0 — ограничитель для выталкивания операций. Закрывающая скобка выталкивает все операции до ближайшего ограничителя и удаляет его из стека. Переопределение открывающей скобки делает невозможным ее использование в прежнем смысле — как знака комментария. Поэтому программисту, вводящему такую надстройку, следует подумать и о решении этого вопроса.

Развивая описанную надстройку дальше, определим простой входной язык с описаниями переменных и присваиваниями, которые записываются обычным образом. В качестве знака присваивания пусть используется слово := , а в качестве разделителя операторов — слово ; . Если переменная использована как получатель присваивания (слева от знака :=), то ее исполнение оставляет на стеке адрес значения; а если данная пере-

менная входит в правую часть присваивания, то ее исполнение кладет на стек само значение данной переменной. Для управления поведением переменных нашего языка введем рабочую переменную ?ЗНАЧ, которая имеет значение 0 при обработке левой части присваивания и значение —1 при обработке правой, и определим слово ПЕРЕМ для описания переменных нашего языка:

```
VARIABLE ?ЗНАЧ
: ПЕРЕМ CREATE 0 , DOES> [COMPILE] LITERAL
 ?ЗНАЧ @ IF STATE @ IF COMPILE @ ELSE @ THEN
                              THEN ; IMMEDIATE
```

Для записи присваиваний определим слова :— и ; через уже определенные скобки:

```
: := [COMPILE] ( -1 ?ЗНАЧ ! ; IMMEDIATE
: ; [COMPILE] ) STATE @ IF COMPILE SWAP COMPILE !
     ELSE SWAP ! THEN  0 ?ЗНАЧ ! ;  IMMEDIATE
```

Слово := кладет на стек ОПРЦ ограничитель для выталкивания операций и устанавливает переменную ?ЗНАЧ на обработку правой части присваивания. Слово ; выталкивает *со* стека ОПРЦ все накопившиеся там операции, в результате на вершине стека данных оказывается значение правой части. Непосредственно под ним располагается адрес переменной, оставленный левой частью данного присваивания. Слова SWAP и ! выполняют присваивание, причем в состоянии компиляции они компилируются, а состоянии исполнения исполняются. В заключение переменная ?ЗНАЧ переустанавливается на режим обработки левой части следующего присваивания.

Благодаря словам := и ; отпадает необходимость в дополнительных внешних скобках для всего выражения. Входной текст в описанном расширении выглядит вполне традиционно:

```
ПЕРЕМ A  ПЕРЕМ B
A := 10 ;   B := 15 ;
A := ( A + B ) * ( A - B ) + 2 ;
```

и вместе с тем это текст на языке Форт! Такие операторы присваивания можно исполнять непосредственно или

включать в тело определений через двоеточие. Однако переопределение точки с запятой усложняет написание новых определений в таком расширении. Чтобы решить эту проблему, можно предусмотреть специальные средства для определения новых слов, как и слова для включения комментариев. Описанный пример еще раз показывает, что средства, которые язык Форт предоставляет программисту, позволяют ему реализовать практически любые необходимые инструментальные надстройки, исходя из конкретных требований и особенностей данной задачи и пожеланий пользователей, с ней работающих.

3.3. Локальные переменные

Стек данных как универсальное средство для передачи параметров и результатов между форт-словами имеет неоспоримые преимущества. Вместе с тем внутри определения он используется для промежуточных вычислений и размещения значений, которые в них участвуют. Это вызывает определенные трудности для доступа к такому локальному значению, поскольку его положение относительно вершины стека постоянно меняется.

Для упрощения работы желательно закрепить за локальным объектами внутри определения некоторые постоянные имена, через которые и осуществлять доступ к ним.

Имеющийся в языке механизм описания переменных в данном случае не подходит, поскольку создает глобальные имена, тогда как требуется именовать локальные объекты, учитывая при этом возможность рекурсивных вызовов. Поставленную задачу решает включение в работу дополнительного стека, отличного от стека данных. Локальные значения размещаются в этом стеке при входе в определение и убираются из него при выходе.

На все время исполнения определения их положение относительно вершины стека остается постоянным, это позволяет организовать очень простой доступ к таким значениям.

Простейшая надстройка над языком Форт, которая позволяет работать с локальными переменными, выглядит так:

```
100 ALLOT  HERE CONSTANT LP0 ( НАЧАЛО ЛОК.СТЕКА)
VARIABLE LP  ( ТЕКУЩАЯ ВЕРШИНА ЛОКАЛЬНОГО СТЕКА)
: INIT ( ->)  LP0 LP ! ;  INIT
: LOC ( N:СЧЕТЧИК->) 1+ 2* LP @ OVER - DUP LP ! ! ;
: UNLOC ( ->)  LP @ @ LP +! ;
: @@ ( N:СМЕЩ->) CREATE , DOES> ( PFA->A) @ LP @ + @ ;
: !! ( N:СМЕЩ->) CREATE , DOES> ( A,PFA->) @ LP @ + ! ;
2 @@ @1  4 @@ @2  6 @@ @3  8 @@ @4  10 @@ @5 ( И Т.Д.)
2 !! !1  4 !! !2  6 !! !3  8 !! !4  10 !! !5 ( И Т.Д.)
```

Вначале отводится область объемом 100 байт и адрес ее конца запоминается как константа LP0. Эта область будет использоваться как локальный стек, растущий в сторону убывания адресов. Переменная LP хранит указатель на текущую вершину локального стека, ее инициализацию выполняет слово INIT, которое присваивает ей значение LP0. Слово LOC резервирует в этом стеке память на N двухбайтных значений, дополнительно отводя еще одну ячейку, в которую засылает значение N — длину всей области. Обратное действие — освобождение памяти — выполняет слово UNLOC , которое использует сохраненное значение N. Слова @@ и !! являются определяющими для двух рядов слов, выполняющих разыменование локальных объектов и присваивание им нового значения. Каждый локальный объект в пределах одного определения идентифицируется своим номером, который включен в имя операций разыменования и присваивания для работы с ним. В качестве примера рассмотрим определение слова 5INV, переставляющего 5 верхних элементов стека данных в обратном порядке:

```
: 5INV ( A,B,C,D,E->E,D,C,B,A)  5 LOC
  !1 !2 !3 !4 !5  @1 @2 @3 @4 @5  UNLOC ;
```

Приведенная реализация локальных переменных предельно упрощена. Ее очевидным усовершенствованием является, например, введение контроля за переполнением и исчерпанием локального стека и включение действий LOC и UNLOC в семантику входа в высокоуровневое определение и выхода из него; при этом подсчет числа локальных объектов может выполняться автоматически, аналогично определению адреса перехода в структурах управления.

Можно пойти еще дальше и вместо прямой нумерации локальных объектов ввести их настоящее описание через соответствующее слово внутри определения через двоеточие с автоматическим переносом входных пара-

метров на локальный стек при входе в такое определение. Словарная статья для имени локального объекта, которая строится по такому описанию, должна располагаться несколько выше текущей вершины словаря и иметь признак немедленного исполнения. Ее действие состоит в компиляции обращения к соответствующему элементу локального стека. По завершении компиляции данного определения все такие временные статьи автоматически исключаются из словаря.

Введение локальных переменных не только упрощает программирование, освобождая программиста от необходимости тщательно отслеживать все изменения на стеке, но и сокращает размер скомпилированной программы, поскольку из нее исчезают сложные последовательности из слов OVER , DUP , ROT , PICK и других, обычно используемых для доступа к локальным значениям. Такое неожиданное сочетание приятного с полезным — одна из многих удивительных сторон языка Форт, которые открываются программисту при знакомстве с ним.

3.4. Векторное поле кода

В языке Форт с каждым словом-командой связано некоторое действие, определяющее семантику данного слова. В словарной статье, которая является внутренним представлением форт-слова, это действие задано через поле кода. В случае косвенного шитого кода оно содержит адрес машинной программы, исполнение которой и составляет действие данного слова. Эта программа образует исполняющую часть определяющего слова, через которое данное слово было создано, и параметризуется адресом поля параметров его словарной статьи.

Описанный механизм является чрезвычайно мощным и в то же время очень компактным в реализации. Вместе с тем и он может быть улучшен с учетом конкретной цели. Заметим, что этот механизм предписывает исполнение одной и той же программы для всех случаев применения данного слова, в то время как в каждом случае слово употребляется в определенном контексте, и его исполнение, вообще говоря, зависит от этого контекста. Разумеется, анализ контекста можно включить в программу слова, если закодировать контекст

через дополнительный параметр или глабальную переменную.

Именно в этом, например, заключается действие стандартного слова LITERAL, которое анализирует текущее состояние текстового интерпретатора через значение переменной STATE . Если это значение —• нуль (состояние исполнения), то это слово больше ничего не делает. При ненулевом значении (состояние компиляции) оно снимает значение с вершины стека и компилирует его как литерал на вершину словаря.

Пример другого подхода дает описанная в п. 3.2 реализация оператора присваивания в традиционной инфиксной форме записи. В зависимости от текущего значения переменной ?ЗНАЧ, которое изменяется словами :=и ;, слова, обозначающие переменные, оставляют на стеке либо адрес, либо значение данной переменной.

Предлагаемое улучшение связано прежде всего с использованием переменных. В стандартном определении при исполнении переменной на стеке будет оставлен адрес ячейки (поля параметров), в которой хранится ее значение. Однако сам по себе этот адрес редко бывает нужен, обычно он используется либо для получения текущего значения переменной с помощью слова @ , либо для засылки в нее нового значения словом ! . Поэтому можно считать, что с переменной связано не одно действие, а три — получение текущего значения, засылка нового и получение адреса. Какое именно действие требуется в каждом конкретном случае, определяется контекстом.

Введем слово QUAN (от QUANTITY — величина), которое определяет новое слово с тремя описанными выше действиями. В словарной статье таких слов вместо одного поля кода создается три — по одному на каждое действие. Будем обозначать их адреса через 0CFA, 1CFA и 2CFA соответственно (рис. 3.1). За ними располагается ячейка, отведенная под текущее значение данной переменной, обозначим ее адрес через 0PFA. Если рассматривать такую структуру как обычную словарную статью, то поле 0CFA является полем кода, а поля 1CFA,2CFA и 0PFA занимают поле параметров. Если в шитый код скомпилирован адрес 0CFA , то при исполнении соответствующего кода в качестве адреса поля параметров выступает адрес 1CFA . Анало-

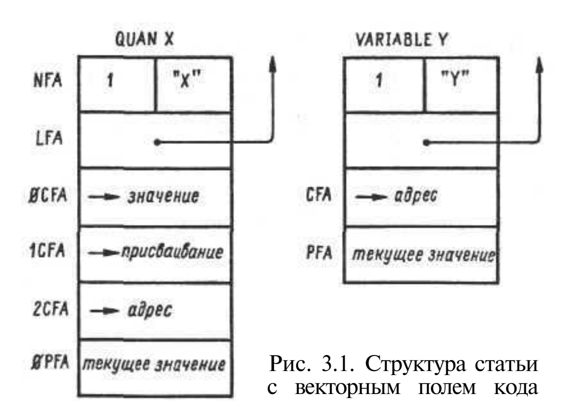


Рис. 3.1. Структура статьи с векторным полем кода

гично для адреса 1CFA полем параметров служит 2CFA , а для 2CFA — адрес 0PFA . Поэтому описанные выше три действия можно задать так:

```
: ЗНАЧ  DOES> ( 1CFA->N:ЗНАЧ) 4 + @ ;
: ПРИСВ DOES> ( N:ЗНАЧ,2CFA->) 2+ ! ;
: АДР   DOES> ( 0PFA->0PFA )        ;
```

Нетрудно увидеть, что действие АДР совпадает со стандартным действием для переменной, состоящим в том, что адрес поля параметров кладется на стек. Определим теперь слово QUAN , используя слово ПРИСВ в качестве вспомогательного:

```
: QUAN ( ->) CREATE LATEST NAME> DUP @ ( КОД 2CFA)
   ПРИСВ SWAP @ ( КОД 2CFA,КОД 1CFA) , , 0 ,
   DOES> ( КОД ДЛЯ 0CFA) 4 + @ ;
```

Создающая часть этого определения использует поле кода создаваемой статьи как рабочую ячейку, из которой сначала извлекается значение для 2CFA , засланное туда словом CREATE, и затем значение для 1CFA , которое засылается туда словом ПРИСВ . Окончательное значение в этой ячейке устанавливается словом DOES> . Описание переменной через слово QUAN выглядит так же, как описание обычной переменной: QUAN X . Однако теперь при исполнении слова X на стеке будет оставлено текущее значение переменной из ее поля 0PFA . Выполнение двух других действий задают слова TO (предлог «в») и AT (предлог «из»)

```
: TO   2+ STATE @ IF , ELSE EXECUTE THEN ;
                                     IMMEDIATE
: AT   4 + STATE @ IF , ELSE EXECUTE THEN ;
                                     IMMEDIATE
```

Эти слова имеют признак немедленного исполнения и в состоянии компиляции (внутри определения через двоеточие) компилируют коды 1CFA и 2CFA для следующего слова. В состоянии исполнения соответствующие действия исполняются. Теперь, чтобы присвоить переменной X значение 100, нужно выполнить текст 100 TO X, а для получения адреса — текст AT X.

Такое усовершенствование, как и введение локальных переменных, не только упрощает программирование, но и сокращает объем скомпилированного кода. Из текста программ исчезает слово @, применявшееся для разыменования адреса переменной. В результате вместо двух ссылок — на поле CFA для переменной и статью для @ — в шитом коде присутствует только одна ссылка (на поле 0CFA). Аналогично в случае присваивания вместо двух ссылок (на поле CFA переменной и на статью для !) компилируется одна (на поле 1CFA). В практических реализациях программы для действий ЗНАЧ и ПРИСВ обычно задаются не в виде высокоуровневых определений, а непосредственно в машинном коде данной ЭВМ через встроенный ассемблер форт-системы. В этом случае описание переменных через слово QUAN не только сокращает размер программы, но и повышает ее быстродействие, поскольку отпадает необходимость в исполнении действия NEXT для интерпретации еще одной ссылки.

По аналогии со словом QUAN определим слово VECT (от VECTOR — вектор), которое также создает словарную статью с векторным полем кода из трех элементов:

```
: VECT ( ->) 0 CONSTANT
         LATEST NAME> DUP @ , ( КОД 2CFA)
  ПРИСВ DUP @ (  1CFA) SWAP >BODY !  ['] ABORT ,
  DOES> ( КОД ДЛЯ 0CFA)  4 + @ EXECUTE  ;
```

Код для поля 2CFA указывает на исполняющую часть из определяющего слова CONSTANT , поле 1CFA такое же, как и для слова QUAN, оно выполняет присваивание нового значения. Наконец, поле 0CFA,

которое задается исполняющей частью определения VECT, исполняет слово, адрес поля кода которого является текущим значением поля 0PFA . Для получения текущего значения и засылки нового по-прежнему используются определенные ранее слова AT и TO. Вот пример на использование этих слов: VECT V ' DUP TO V . Теперь исполнение слова V равносильно исполнению DUP. Таким образом, слова, определенные через VECT , можно рассматривать как переменные, значениями которых являются другие слова. Для исполнения слова — текущего значения такой переменной — достаточно указать только имя этой переменной без дополнительных операций @ и EXECUTE. Текст AT V> NAME ID. распечатывает имя слова — текущего значения V . Разумеется, при такой распечатке предполагается, что само это слово имеет обычную структуру словарной статьи.

3.5. Выбор по целому

Выбор по целому — распространенная конструкция в языках программирования. Она является обобщением условного оператора, который осуществляет выбор между двумя последовательностями операторов — частью «то» и частью «иначе» — по логическому значению (ИСТИНА или ЛОЖЬ) условия. В конструкции выбора по целому в качестве значения условия выступает целое число, и выбор осуществляется между несколькими альтернативными ветвями, каждая из которых соответствует определенному значению условия или некоторому множеству таких значений. Как правило, множества значений условия для разных ветвей не должны пересекаться. Обычно эти множества задают явным перечислением отдельных значений или указанием диапазона для них. Разберем два варианта реализации выбора по целому. В первом используется переключатель, во втором — вложенные условные операторы.

В случае реализации *через переключатель* каждая ветвь должна задаваться отдельным форт-словом. Пусть имеется несколько ветвей, каждая из которых задается своим порядковым номером. Из адресов компиляции этих ветвей в соответствии с их порядковыми номерами строится вектор-переключатель, и нужная

ветвь выбирается по порядковому номеру. Определяющее слово SWITCH (переключатель) компилирует такой вектор в поле параметров определяемого слова и задает выбор альтернативы в исполняющей части:

```
: SWITCH ( ->) ?EXEC CREATE SMUDGE ]
   DOES> ( I:НОМЕР ВЕТВИ,PFA->) SWAP
                        1- 2 * + @ EXECUTE ;
```

Создающая часть строит новую словарную статью и переключает текстовый интерпретатор в состояние компиляции. Таким образом, следующие слова будут компилироваться в поле параметров определяемого слова. Слово SMUDGE выставляет разряд «Не готов» в байте-счетчике поля имени, делая данную статью «невидимой» при поиске слов в словаре. Этот разряд будет сброшен словом ; (точка с запятой), которое завершает компиляцию такого переключателя. Например:

```
: W1 ." ПОНЕДЕЛЬНИК" ;  : W2 ." ВТОРНИК" ;
: W3 ." СРЕДА" ;        : W4 ." ЧЕТВЕРГ" ;
: W5 ." ПЯТНИЦА" ;      : W6 ." СУББОТА" ;
: W7 ." ВОСКРЕСЕНЬЕ" ;
SWITCH ДЕНЬ-НЕДЕЛИ  W1 W2 W3 W4 W5 W6 W7 ;
```

Порядок использования переключателя ДЕНЬ-НЕДЕЛИ иллюстрирует следующий протокол работы:

```
> 3  ДЕНЬ-НЕДЕЛИ
СРЕДА  OK
> 5  ДЕНЬ-НЕДЕЛИ
ПЯТНИЦА  OK
```

Описанный механизм аналогичен определению вектора, рассмотренному в п. 1.10. Его также можно задавать по-разному в зависимости от конкретных требований. В приведенной реализации неправильное значение выбирающего условия (выходящее за диапазон от 1 до 7) приведет к непредсказуемому результату, поскольку соответствующий контроль отсутствует. Чтобы ввести его в реализацию, нужно каким-то образом подсчитать число скомпилированных ссылок при завершении компиляции. Для этого можно применить способ, аналогичный компиляции переходов в шитом коде, введя вместо точки с запятой специальное слово, отмечающее конец переключателя.

Для реализации конструкции выбора *через вложенные условные операторы* создаются специальные слова, которые обрамляют всю конструкцию и каждую ее ветвь. В этом случае ветвь является обычной последовательностью форт-слов, аналогичной части «то» или части «иначе» условного оператора. Как и в случае условного оператора, конструкцию выбора можно использовать только внутри определений через двоеточие, т. е. в состоянии компиляции текстового интерпретатора. При этом все ее специальные слова имеют признак немедленного исполнения и компилируют необходимые проверки и переходы.

```
: CASE ( ->A0:CSP,4)  ?COMP CSP @ !CSP 4 ;
                                     IMMEDIATE
: OF ( 4->A1:>MARK,1,5) 4 ?PAIRS COMPILE OVER
  COMPILE = [COMPILE] IF  COMPILE DROP 5 ;
                                     IMMEDIATE
: ENDOF ( A1:>MARK,1,5->A2:>MARK,1,4)   5 ?PAIRS
     [COMPILE] ELSE  4 ;             IMMEDIATE
: ENDCASE ( A0,A1,1,...,AN,14->)
  4 ?PAIRS COMPILE DROP
  BEGIN SP@  CSP @ = 0= WHILE  [COMPILE] THEN
  REPEAT ( A0) CSP ! ;               IMMEDIATE
: ДЕНЬ-НЕДЕЛИ ( N->)  CASE
1 OF ." ПОНЕДЕЛЬНИК" ENDOF 2 OF ." ВТОРНИК" ENDOF
3 OF ." СРЕДА" ENDOF  4 OF ." ЧЕТВЕРГ" ENDOF
5 OF ." ПЯТНИЦА" ENDOF  6 OF ." СУББОТА" ENDOF
7 OF ." ВОСКРЕСЕНЬЕ" ENDOF
  CR . ." - ДЕНЬ НЕДЕЛИ?" ABORT ENDCASE ;
```

Слова CASE (выбор) и ENDCASE (конец выбора) ограничивают конструкцию и обеспечивают правильную компиляцию вложенных операторов. Слово CASE , проверив, что текстовый интерпретатор находится в состоянии компиляции, сменяет глобальную переменную CSP (сокращение от CURRENT STACK POINTER — текущий указатель стека), сохраняя на стеке ее прежнее значение (слово !CSP засылает в переменную CSP адрес текущей вершины стека). Слова OF (из) и ENDOF (конец из) ограничивают отдельную ветвь. Во время работы скомпилированного определения перед началом исполнения каждой ветви на стеке лежат два значения: число, представляющее условие выбора, и номер данной ветви. Слово OF компилирует текст OVER =

= IF DROP , который обеспечивает передачу управления на данную ветвь, если эти два значения совпали, причем в этом случае они оба снимаются со стека. Если же значения оказались разными, то управление передается на текст, следующий за словом ENDOF для данной ветви, которое эквивалентно слову ELSE . Наконец, слово ENDCASE компилирует операцию DROP , чтобы снять со стека оставшееся там значение условия, и разрешает все накопившиеся на стеке выходы из ветвей на текущую точку, исполняя для этих ветвей слово THEN . Его последним действием является восстановление прежнего значения переменной CSP , которое к этому моменту оказывается на вершине стека.

В приведенных определениях можно несколько уменьшить объем кода, компилируемого для входа в каждую ветвь, если ввести для этого специальное слово, вслед за которым в шитый код компилируется адрес перехода на следующую ветвь:

```
: (OF) ( N:УСЛОВИЕ,I:НОМЕР ВЕТВИ->)
    OVER = IF DROP R> 2+ ELSE R> @ THEN  >R ;
: OF  ( 4->A:>MARK,2,5)  4 ?PAIRS
      COMPILE (OF) >MARK  2  5 ;     IMMEDIATE
```

Исполнение слова (OF) аналогично ?BRANCH , в зависимости от условия оно переустанавливает указатель интерпретации, либо обходя скомпилированную за ним ссылку, либо устанавливая указатель по значению этой ссылки. Аналогичным образом наряду со словами (OF) и ОФ можно определить пары (<OF) и <OF, (> OF) и > OF , (<OF<) и <OF< , выполняющие сравнение значения условия с заданным значением, выбирающим данную ветвь, не на равенство, а на неравенство указанного вида. При этом слово (<OF<) сравнивает значение условия с двумя значениями, определяющими интервал. Например:

```
: ПРИЕМ ( N:НОМЕР ДНЯ->)   CASE
    3 OF ." НЕПРИЕМНЫЙ" ENDOF
  1 5 <OF< ." ПРИЕМНЫЙ" ENDOF
  6 7 <OF< ." ВЫХОДНОЙ" ENDOF
CR . ." - НОМЕР ДНЯ?" ABORT ENDCASE  ." ДЕНЬ" ;
```

В данном примере множества выбирающих значений пересекаются, и выбор ветви определяется для этого

случая порядком расположения ветвей в **конструкции выбора.**

3.6. Динамическая идентификация

Внутренним представлением форт-слова является словарная статья, которая размещается в словаре. Через поля связи словарные статьи объединяются в цепные списки, каждый из которых также представлен как отдельный объект словарной статьей, создаваемой по слову VOCABULARY . Исполнение слова, обозначающего такой список, делает его текущим значением переменной CONTEXT , определяющей список, в котором начинается поиск каждого вводимого форт-слова. Если слово отсутствует в этом списке, то следующим просматривается список, в который добавляются создаваемые новые статьи, текущее значение переменной CURRENT . Последним просматривается стандартный список FORTH . Если текстовый интерпретатор находится в состоянии исполнения, то найденный код исполняется, в состоянии компиляции он компилируется в виде ссылки на данную словарную статью для последующего исполнения в составе скомпилированного определения.

Таким образом, в процессе создания нового определения мы индентифицируем составляющие его слова *статически,* т. е. по текущему контексту (значениям переменных CONTEXT и CURRENT) во время компиляции. Во время исполнения этого определения будут исполняться именно эти, найденные во время компиляции составляющие слова. Вместе с тем в практике программирования уже давно применяется прием, известный как *динамическая идентификация. В* применении к языку Форт он состоит в том, что вместо компиляции ссылки на статью слова по результату статической индентификации в компилируемой статье запоминается имя слова, и его поиск ведется в момент фактического исполнения. Главная особенность состоит в том, что контекст для поиска также устанавливается динамически в результате предшествующих вычислений.

Описанный подход позволяет в ряде случаев существенно упростить программирование слов, исполнение которых зависит от контекста их вызова. **Вместо того,**

чтобы внутри такого слова анализировать контекст и в зависимости от него выполнять те или иные действия, можно создать ряд слов с одним и тем же именем, но в разных контекстах. Каждое такое слово выполняет только свое действие, а вместо анализа контекста в месте вызова такого слова программируются установка нужного контекста и поиск слова по заданному имени с последующим исполнением кода.

В качестве примера рассмотрим простой язык, аналогичный рассмотренному в п. 3.2, и содержащий описания констант, переменных и оператор присваивания с инфиксной формой записи для выражений. Пусть описания констант выглядят, например, так: КОНСТ А = 2. Чтобы не перегружать наш пример лишними деталями, не будем учитывать возможность компиляции описаний и операторов этого языка. Пусть они сразу же исполняются по мере поступления. Собственно задача состоит в том, что при обработке левой части оператора присваивания можно было различать переменные и константы. В первом случае результатом исполнения должен быть адрес соответствующей ячейки памяти, а во втором — сообщение об ошибке — недопустимом использовании константы в качестве получателя присваивания. Если же имя переменной или константы использовано в правой части присваивания, то в обоих случаях нужно получить соответствующее значение. Разумеется, соответствующие операции проверки можно включить в программу для переменной и константы, анализируя глобальную переменную ?ЗНАЧ , как это сделано в рассмотренном выше примере. Покажем, как эту же задачу можно решить через динамическую идентификацию.

Определим два контекста (списка слов) П и К, содержащих операции над переменными и константами. Таких операций две: ЗНАЧ для получения значения и АДР для получения адреса. Операция ЗНАЧ работает в обоих случаях одинаково, а операция АДР допустима только для переменной; будучи примененной к константе, она должна выдать сообщение об ошибке. В список FORTH также включим слово ЗНАЧ , которое в этом контексте выполняет пустую операцию, и слово АДР, выдающее сообщение об ошибке (их назначение станет ясным чуть позже). Для определения всех этих слов примем, что значение переменной или константы

размещается в поле параметров созданной для нее словарной статьи.

```
FORTH DEFINITIONS
VOCABULARY П   VOCABULARY K
П DEFINITIONS ( ДЕЙСТВИЯ С ПЕРЕМЕННЫМИ)
: ЗНАЧ ( PFA->N:ЗНАЧ)  @ FORTH ;
: АДР  ( PFA->A:АДР)  FORTH ;
K DEFINITIONS  ( ДЕЙСТВИЯ С КОНСТАНТАМИ)
: ЗНАЧ ( PFA->N:ЗНАЧ)  @ FORTH ;
: АДР  ( PFA->) CR
  ." НЕДОПУСТИМОЕ ИСПОЛЬЗОВАНИЕ КОНСТАНТЫ "
    BODY> >NAME ID. ABORT ;
FORTH DEFINITIONS
: ЗНАЧ ( ->)  ;
: АДР  ( ->) -1 ABORT" НЕДОПУСТИМЫЙ АДРЕС" ;
```

Заметим, что при исполнении слов ЗНАЧ и АДР в контекстах П и К помимо вычисления соответствующего значения текущий контекст переключается на FORTH . Исполняющая часть определяющих слов ПЕРЕМ и КОНСТ теперь состоит в установке соответствующего контекста, при этом на стек кладется адрес поля параметров данной статьи:

```
: ПЕРЕМ ( ->) CREATE 0 , DOES> ( PFA->PFA) П ;
: КОНСТ ( ->) CREATE BL WORD DROP
     ( ВЫБРАТЬ ЗНАК =)
    BL WORD NUMBER DROP ( N:ЗНАЧ) ,
            DOES> ( PFA->PFA) K ;
```

Чтобы теперь при обработке выражения в левой части присваивания получать значение переменной или константы, надо слегка изменить определение слова)ОПРЦ) (см. п. 3.2) — включить в него исполнение операции ЗНАЧ, которая идентифицируется динамически по текущему контексту.

```
: >ОПРЦ> ( N:ПРИОРИТЕТ->)
           >R " ЗНАЧ" FIND DROP EXECUTE
    BEGIN ОПРЦ@ R@ < NOT WHILE  ОПРЦ>  DROP
      ОПРЦ> ( CFA) EXECUTE REPEAT R> DROP ;
```

Если это определение исполняется сразу после переменной или константы (в контексте П или К), то оно преобразует адрес поля параметров соответствующей статьи, который находится в этот момент на вершине

стека, в значение данной переменной или константы. Для всех других случаев (контекст FORTH) преобразования стека не происходит. Именно по этой причине определения ЗНАЧ в списках П и К по завершении их исполнения переключают текущий контекст на FORTH , а в списке FORTH присутствует определение ЗНАЧ с пустым действием. Последнее обстоятельство гарантирует успешный поиск слова ЗНАЧ в любом контексте, поэтому в приведенном определении проверка того, что поиск закончился успешно, опущена (is DROPped). В заключение осталось только аналогичным образом переопределить слово : = (см. п. 3.2):

```
: := " АДР" FIND DROP EXECUTE [COMPILE] ( ;
```

Сравнивая получившиеся определения, мы видим, что они существенно сократились в объеме: из них исчезли анализ текущего контекста и (соответственно) условные операторы, предписывающие выполнение разных действий в зависимости от результата анализа. Описанный подход особенно перспективен, когда требуется выполнить много разных операций над объектами, каждая из которых выполняется по-разному в зависимости от текущего контекста. Разбиение большой операции с множеством проверок и разветвлений на ряд мелких, каждая из которых выполняется в своем контексте, не только упрощает программирование и отладку, но и сокращает объем программы, так как эти мелкие специализированные операции для разных объектов часто можно совмещать:

```
K DEFINITIONS
: ЗНАЧ   ( PFA->N:ЗНАЧ)  [ П ]  ЗНАЧ  ;
```

В приведенном определении слово ЗНАЧ для списка К не программируется вновь, а определяется через такое же слово из контекста П , тем самым сокращается общий объем отладки. Следуя по этому пути, можно вообще исключить слово АДР из списка К , поскольку в этом случае сообщение об ошибке выдаст слово АДР из списка FORTH , правда, с меньшей информацией — не будет напечатано имя константы.

Если механизм динамической идентификации применяется для ряда не связанных между собой целей, то может оказаться неудобным выполнять переключение контекстов через одну и ту же глобальную пере-

менную CONTEXT и выполнять поиск слова в текущем контексте словом FIND. Как правило, все форт-системы имеют более элементарное слово (FIND), которое получает в качестве параметра текстовую строку с именем слова и вход в список статей, в котором нужно выполнять поиск. Используя это слово, можно организовать поиск слова в любом списке или группе списков, уже не связывая его с текущим порядком поиска, принятым для слова FIND.

Например, именно таким образом нетрудно ввести в язык обработку исключительных ситуаций. Для этого нужно завести стек, элементами которого являются списки слов для обработки ситуаций. При установке перехвата ситуаций на вершину этого стека добавляется новый элемент — список слов-обработчиков. При снятии перехвата верхний элемент снимается со стека, делая доступными другие слова-обработчики с теми же именами. Слово, возбуждающее исключительную ситуацию, обращается к (FIND) для просмотра всех списков в этом стеке от верхнего элемента к самому нижнему, сообщая ему имя слова, обозначающего ситуацию. Первое найденное при таком поиске слово и будет использовано как обработчик ситуации. Если слово отсутствует во всех списках, то это ошибка — непредусмотренная ситуация.

3.7. Многозадачный режим

Запуская на исполнение какое-либо форт-слово, программист имеет возможность исполнить следующее только по завершении исполнения предыдущего. Если исполнение данного слова требует значительного времени счета, то длительные перерывы в диалоге создают определенное неудобство. Ввиду этого представляется весьма привлекательной идея *многозадачной форт-системы,* в которой программист может создавать фоновые задачи и запускать их на исполнение, не прерывая диалога. Текстовый интерпретатор такой системы, обеспечивающий диалог, является одной из задач и разделяет центральный процессор наряду с другими задачами. Идея многозадачного режима привлекательна еще и тем, что разные задачи могут разделять значительную часть кода — практически все ядро форт-системы. Из стандартных слов только системные пере-

менные (STATE , BASE и другие) и списки слов (FORTH , ASSEMBLER) не могут одновременно участвовать в нескольких вычислительных процессах (задачах), поскольку их код (значение поля параметров) в процессе работы может изменяться. В то же время вся остальная часть кода форт-системы, включая все слова, определенные через двоеточие и CONSTANT, в процессе работы не меняется и может даже размещаться в постоянной оперативной памяти (ПЗУ). Чтобы снять это ограничение, во многих реализациях выделены особая область для размещения значений, которые могут изменяться во время счета. Эта область называется *пользовательской* и обычно располагается в конце адресного форт-пространства рядом со стеками и буферным пулом. Распределение памяти внутри пользовательской области выполняет сам программист через определяющее слово USER (пользовательский):

```
: USER ( N:СМЕЩЕНИЕ-> )  CREATE  ,
        DOES> ( PFA->A:АДРЕС)  @  U0 + ;
```

По характеру использования это слово аналогично CONSTANT ; значение, которое оно снимает со стека и компилирует в поле параметров определяемого слова, представляет собой смещение от начала пользовательской области. При исполнении такого слова на стеке будет оставлен соответствующий адрес (слово U0, использованное в определении слова USER, дает адрес начала пользовательской области). При наличии пользовательской области все системные переменные размещаются в ней, занимая какую-то ее начальную часть. Их словарные статьи определяются через слово USER, тем самым обеспечивается их. неизменяемость в процессе работы.

Во всех реализациях многозадачного режима с разделяемым общим ядром форт-системы каждая задача имеет собственные пользовательскую область, стек данных и стек возвратов. Текущее состояние задачи определяется значением указателя интерпретации и адресами вершин этих стеков. Представлением самой задачи обычно служит адрес начала ее пользовательской области, в которой предусматривается память для сохранения текущего состояния на то время, пока задача неактивна. Для возобновления исполнения за-

дачи требуется восстановить ее текущее состояние из области сохранения и передать управление на точку NEXT адресного интерпретатора.

Дальнейшие уточнения конкретного варианта многозадачного режима зависят от многих частных причин. Для примера рассмотрим реализацию системы ПОЛИФОРТ фирмы «Форт» [31]. Эта система реализована для целого ряда ЭВМ, включая персональный компьютер ИБМ. Использованный в ней механизм переключения задач основан на кольцевом принципе: все задачи связаны в кольцо через начальную часть своей пользовательской области (рис. 3.2). Задачи, составляющие кольцо, по очереди получают центральный процессор и удерживают его до тех пор, пока не исполнят слово PAUSE (пауза) или STOP (стоп). Для определений в машинном коде предусмотрен аналогичный код WAIT (ждать). Многие слова, которые выполняют асинхронные операции обмена (TYPE , EXPECT , BLOCK , BUFFER), содержат код WAIT или переход на STOP ,

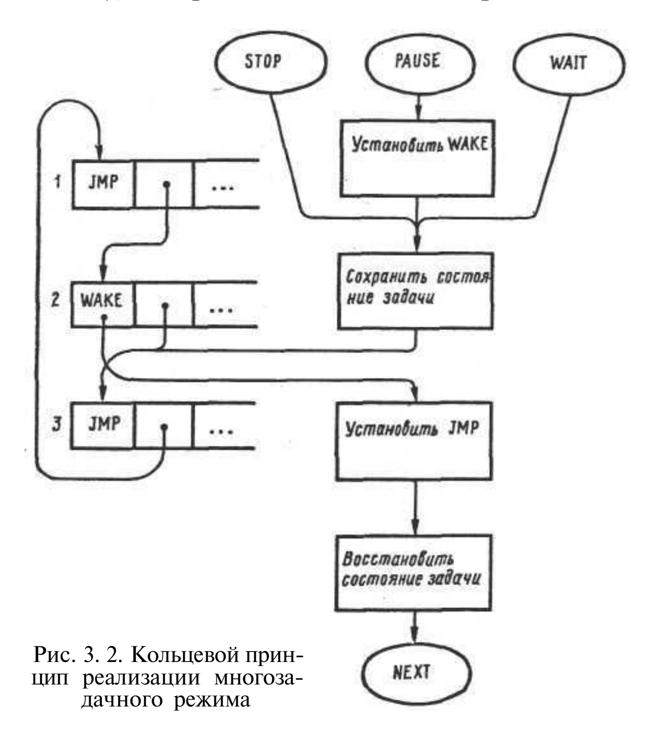


Рис. 3. 2. Кольцевой принцип реализации многозадачного режима

чтобы в то время, когда они ждут завершения обмена, другие задачи могли использовать центральный процессор. Задачи, в которых для выполнения длительных вычислений нет необходимости обращаться к операциям обмена, должны сами периодически предоставлять процессор другим задачам, выполняя слово PAUSE.

Все задачи, существующие в системе, связываются в кольцо через команду безусловного перехода (JMP), которая располагается в начале их пользовательских областей. В качестве звена связи используется адресная часть команды перехода, указывающая на такую же команду следующей задачи в кольце. Если передать управление на одну из таких команд, то будет исполняться бесконечный цикл из переходов по кольцу задач до тех пор, пока какая-нибудь задача не будет разбужена.

Для пробуждения задач часто используются асинхронные подпрограммы обработки прерываний. Например, если выход по прерыванию, установленный в слове EXPECT , распознает код возврата каретки, которым заканчивается вводимая строка, то он будит задачу, запросившую данный ввод с терминала, засылая команду WAKE (разбудить) на место ее команды JMP . Для этого часто используется какая-нибудь команда перехвата (например, для ЭВМ СМ-4 это команды TRAP и EMT). Когда цикл передач управления по кольцу задач дойдет до этой команды, она передаст управление специальной подпрограмме-побудчику, которая и возобновит исполнение данной задачи от точки последнего останова. При этом на место команды WAKE будет вновь заслана команда JMP, и исполнение данной задачи будет продолжаться до следующей операции PAUSE , STOP или WAIT . Пример цикла ввода и обработки данных с возобновлением по прерыванию дает следующее определение:

```
: СБОР  ( -> ) BEGIN ВВОД STOP ОБРАБОТКА AGAIN ;
```

Пробуждение задачи от точки STOP на каждом повторении цикла выполняет подпрограмма обработки прерывания, связанная с вводом данных, когда данные, получаемые при исполнении слова ВВОД , готовы для обработки. Пример задания регулярных остановов в за-

даче, требующей длительного счета, дает такое определение:

```
: СЧЕТ ( ->) 30000 0 DO ШАГ PAUSE LOOP ;
```

При каждом повторении цикла после исполнения слова ШАГ происходит передача управления в кольцо задач с тем, чтобы дать поработать другим задачам, ожидающим центральный процессор.

Действие STOP (для определений, заданных в машинном коде, аналогичное ему действие WAIT) состоит в сохранении текущего состояния задачи (указателя интерпретации, указателя вершины стека данных и указателя вершины стека возвратов) в ее пользовательской области и передаче управления по адресу из ее звена связи в кольце задач. В результате центральный процессор будет исполнять цикл из передач управления по кольцу задач, пока не встретит команду WAKE. Действие PAUSE отличается от описанного действия STOP только тем, что предварительно засылает команду WAKE в начало области данной задачи, обеспечивая тем самым пробуждение данной задачи после полного круга передач управления. Действие WAKE заключается в том, что команда JMP засылается на отведенное ей место в начало пользовательской области, восстанавливается текущее состояние данной задачи из ее области сохранения, управление передается на точку NEXT адресного интерпретатора. В результате исполнение задачи возобновляется от точки останова.

Для определения *фоновой задачи* используется определяющее слово BACKGROUND (фон), которое снимает со стека три числа: размер пользовательской области, размер стека данных и размер стека возвратов. Это слово резервирует в словаре память указанного объема и указатели на эти области компилирует в поле параметров создаваемой статьи, например, 40 60 50 BACKGROUND T . При исполнении слова T на стеке оставляется адрес его поля параметров, через который можно добраться до пользовательской области и стеков задачи T . Данное описание только подготавливает задачу, но не включает ее в кольцо задач и не запускает на исполнение.

При запуске форт-системы в кольце задач присутствует только одна задача — OPERATOR (оператор), звено связи которой указывает на нее же. Для встраи-

вания в кольцо новых задач используется слово BUILD (построить), например T BUILD . Это слово, используя адрес поля параметров статьи для задачи, инициализирует ее пользовательскую область и включает ее в кольцо задач после задачи OPERATOR . При этом в начало области вписывается команда JMP , предохраняющая данную задачу от преждевременного исполнения. Заметим, что форт-слова, составляющие программу данной задачи, могут быть определены значительно позже. Обратное действие — исключение задачи из кольца задач — выполняет аналогичное по употреблению слово RUIN (разрушить).

Для запуска задачи используется слово ACTIVATE (запустить), которое снимает со стека адрес поля параметров статьи для задачи. Оно очищает стек данных и стек возвратов задачи, сбрасывая на начало указатели их вершин в области сохранения, а в качестве точки возобновления засылает свой адрес возврата, указывающий на следующее за ним слово. Поскольку адрес снимается со стека возвратов, то слово ACTIVATE аналогично слову EXIT возвращает управление на точку вызова определения, его вызвавшего. Последним действием слова ACTIVATE является подготовка задачи к пробуждению путем засылки команды WAKE в начало ее пользовательской области. Приведем пример использования слова ACTIVATE и запуска задачи на исполнение:

```
: СЧЕТ ( PFA->) ACTIVATE BEGIN ШАГ PAUSE LOOP ;
T СЧЕТ
```

Задача T будет выполнять бесконечный цикл, разделяя центральный процессор с текстовым интерпретатором форт-системы — задачей OPERATOR . Исполнение задачи можно оставить с помощью слова HALT (останов):

```
: HALT ( PFA-> ) ACTIVATE STOP ;
```

если в рамках задачи OPERATOR исполнить текст T HALT . После такого останова данную задачу можно запустить на счет с какой-нибудь другой форт-программой.

При работе в многозадачном режиме возникает проблема *синхронизации задач,* разделяющих общие ресурсы, которые требуют монопольного использования.

Примерами таких ресурсов являются внешние устройства, общие области оперативной памяти, куда осуществляется запись данных, общие подпрограммы, не допускающие параллельного использования. Опишем простой механизм *семафоров,* позволяющий управлять использованием таких ресурсов в многозадачном режиме.

С каждым ресурсом свяжем переменную — его семафор, которая должна размещаться в общей области форт-системы, доступной для всех задач. Когда ресурс не занят, эта переменная содержит нуль. Когда ресурс занят какой-нибудь задачей, эта переменная содержит адрес пользовательской области данной задачи.

Определим слова GET (получить) и RELEASE (освободить), аналогичные семафорным операциям Дейкстры [8]. В определениях этих слов используется вспомогательное слово FREE (свободен), проверяющее, свободен ли ресурс. Заметим, что ресурс свободен с точки зрения данной задачи, если он не занят или занят этой же задачей.

```
: FREE ( A:СЕМАФОР->A,F:ПРИЗНАК СВОБОДНОСТИ )
      @ DUP 0= SWAP U0 = OR ;
: GET  ( A:СЕМАФОР-> )
      BEGIN PAUSE FREE UNTIL  U0 SWAP ! ;
: RELEASE ( A:СЕМАФОР->)
     FREE IF 0 SWAP ! ELSE DROP THEN ;
```

В приведенных определениях слово U0 возвращает адрес начала пользовательской области текущей задачи. Слово GET проверяет, свободен ли ресурс в каждом цикле передач управления по кольцу задач, и занимает его, как только обнаружит, что ресурс свободен. Слово RELEASE освобождает ресурс, если он был занят данной задачей.

Разумеется, эти слова не обеспечивают защиты от взаимных блокировок задач, каждая из которых ожидает ресурса, который занят другой задачей:

```
VARIABLE ДИСК  VARIABLE ЛЕНТА
: ШАГ1  ДИСК GET  ЛЕНТА GET  ...  ;
: ШАГ2  ЛЕНТА GET  ДИСК GET  ...  ;
```

Если слова ШАГ1 и ШАГ2 исполняются в разных задачах, то это может привести к их взаимной блокировке. Наилучший способ избежать такой ситуации — не

запрашивать более одного ресурса единовременно. Например, в случае работы с диском и лентой, задача может запросить ресурс ДИСК, выполнить слово BLOCK для получения данных, затем переслать эти данные в свою локальную область и, освободив ДИСК, запросить ресурс ЛЕНТА.

3.8. Сопрограммы

Представление задачи в виде взаимодействующих *сопрограмм* (процессов) в теории программирования стало уже традиционным [15,16]. Но на практике этот метод используется редко, так как большинство распространенных языков высокого уровня не имеют необходимых конструкций. В стандарт языка Форт также не включены средства для сопрограммной работы. Вызвано это в основном тем, что такие средства обычно зависят от конкретного приложения и могут варьироваться очень сильно. Тем не менее, используя только стандартные для большинства систем слова, можно реализовать необходимые конструкции.

Рассмотрим достаточно простую и экономную реализацию сопрограмм. Направления, в которых можно ее усовершенствовать, рассмотрены ниже.

Сопрограмма состоит из двух частей — собственной области памяти и программы, работающей с этой областью памяти. Для языка Форт программа — это некоторое форт-слово, а область памяти должна содержать место для указателей стеков, для самих стеков и для служебной информации.

Будем считать, что адрес области памяти текущей сопрограммы находится в переменной Т-С . Используя ее, введем слова для доступа к полям этой области. Для простоты в данной реализации отводится участок памяти размером 128 байт. В его начале находятся: Т-ПРОГ — указатель начала шитого кода, соответствующего исполняемой программе, Т-СТЕК — указатель вершины стека данных на момент последней приостановки данной сопрограммы ' (указатель вершины стека возвратов сохраняется на стеке данных) и Т-ВЫЗВ — адрес аналогичной области для сопрограммы, вызвавшей данную. В остальной памяти участка размещаются стек данных и стек возвратов по 60 байт каждый.

Внешняя программа, в рамках которой действует описываемый механизм сопрограмм, представлена 6-байтной областью ВНЕШ , в которой размещаются перечисленные выше значения. В качестве своих стеков эта сопрограмма использует исходные стеки форт-системы.

Перед началом работы все сопрограммы должны быть проинициализированы словом START (старт), завершение работы сопрограммного механизма вызывает слово STOP (стоп).

Сопрограмма определяется через слово СОПРОГРАММА , которое по своему употреблению аналогично двоеточию. Вслед за ним идет имя сопрограммы и форт-текст, задающий требуемую программу. Обычно она состоит из цикла, внутри которого используется слово RESUME (возобновить) для приостановки данной сопрограммы и возобновления сопрограммы, ее вызвавшей. Выход по EXIT или ; приводит к завершению работы всего сопрограммного механизма и возобновлению текстовой интерпретации.

```
QUAN T-C   ( АДРЕС ОБЛАСТИ ТЕКУЩЕЙ СОПРОГРАММЫ)
: T-ПРОГ ( ->A:АДРЕС НАЧАЛА ПРОГРАММЫ)  T-C ;
: T-СТЕК ( ->A:АДРЕС УКАЗАТЕЛЯ СТЕКА )  T-C 2+ ;
: T-ВЫЗВ ( ->A:АДРЕС ВЫЗВАВШЕЙ СОПРОГРАММЫ)T-C 4 + ;
CREATE ВНЕШ  6 ALLOT  ( ОБЛАСТЬ ДЛЯ ВНЕШНЕЙ ПРОГРАММЫ)
: RESUME ( ->  ОСТАНОВИТЬ ТЕКУЩ.И ВОЗОБН.ВЫЗВАВШУЮ)
 RP@  SP@  T-СТЕК !  ( СОХРАНИТЬ СОСТОЯНИЕ ТЕКУЩЕЙ)
 T-ВЫЗВ @  TO T-C  ( ПЕРЕКЛЮЧИТЬСЯ НА ВЫЗВАВШУЮ ДАННУЮ)
 T-СТЕК @  SP!  RP!  ; ( ВОЗОБНОВИТЬ ПРИОСТАНОВЛЕННУЮ)
: STOP ( -> ЗАВЕРШЕНИЕ ВСЕХ СОПРОГРАММ)  QUIT ;
: (START) ( A:АДРЕС ОБЛАСТИ СОПРОГРАММЫ->  ИНИЦИАЛИЗ)
  DUP TO T-C  T-ПРОГ @ OVER 126 + !  DUP 126 +
  OVER 66 + !  66 + T-СТЕК !  ВНЕШ TO T-C  ;
: СОПРОГРАММА (  ->  ОПРЕДЕЛЕНИЕ СОПРОГРАММЫ) CREATE HERE
   DUP 132 + , 126 ALLOT  HERE 2+ ,  ['] STOP , (START) ]
   DOES>  ( PFA:ОБЛАСТЬ СОПРОГРАММЫ->)  RP@ SWAP >R
   SP@ T-СТЕК !  T-C  R> TO T-C  T-ВЫЗВ !
    T-СТЕК @  SP!  RP!  ;
: START ( -> ИНИЦИАЛИЗИРОВАТЬ СОПРОГРАММУ)
     ' >BODY [COMPILE] LITERAL   STATE @
      IF COMPILE (START) ELSE (START) THEN ;  IMMEDIATE
```

В качестве примера рассмотрим часто встречающуюся задачу: прочитать некоторый файл с одной длиной записи, обработать его и записать в другой файл, причем файлы имеют разную длину записи. Если решать эту задачу без сопрограмм, то надо либо вводить переменные — флажки, сигнализирующие о состоянии буферов, либо найти общее кратное длин входной и выходной записи и, отведя буфера такого размера,

сначала в цикле заполнять входной буфер, а потом его обрабатывать и выводить. В любом случае логически ясные действия по чтению, обработке и записи будут собраны в одну программу, которая в силу этого будет сложна для понимания, отладки и модификации.

Применяя описанный механизм, определим две сопрограммы ВВОД и ВЫВОД , которые обмениваются между собой через однобайтный буфер ТЕК-СИМ , и слово ЗАДАЧА, которое выполняет требуемую перепись данных:

```
QUAN ТЕК-СИМ   ( ОЧЕРЕДНОЙ ОБМЕНИВАЕМЫЙ СИМВОЛ)
CREATE ВХОД-БУФ 80 ALLOT  ( БУФЕР ВВОДА)
CREATE ВЫХ-БУФ  64 ALLOT  ( БУФЕР ВЫВОДА)
СОПРОГРАММА ВВОД
    ОТКРЫТЬ-ВВОД  BEGIN  ВХОД-БУФ ЧИТАТЬ WHILE
    ВХОД-БУФ 80 + ВХОД-БУФ DO  I C@ TO ТЕК-СИМ
                   RESUME LOOP
    REPEAT  ЗАКРЫТЬ-ВВОД  ЗАКРЫТЬ-ВЫВОД ;
СОПРОГРАММА ВЫВОД
    ОТКРЫТЬ-ВЫВОД BEGIN  ВЫХ-БУФ 64 + ВЫХ-БУФ DO
       ТЕК-СИМ I C!  RESUME  LOOP
    ВЫХ-БУФ ПИСАТЬ  AGAIN ;
: ЗАДАЧА
    START ВВОД  START ВЫВОД
    BEGIN ВВОД ВЫВОД AGAIN ;
```

Два ряда слов ОТКРЫТЬ-ВВОД, ЧИТАТЬ, ЗАКРЫТЬ-ВВОД и ОТКРЫТЬ-ВЫВОД , ПИСАТЬ , ЗАКРЫТЬ-ВЫВОД обеспечивают взаимодействие с входным и выходным файлами. Слова ЧИТАТЬ и ПИСАТЬ требуют в качестве параметра адрес буфера (ВХОД-БУФ для ввода и ВЫХ-БУФ для вывода), а слово ЧИТАТЬ , кроме того, возвращает логический результат — признак успешного завершения чтения.

В заключение рассмотрим направления, в которых можно развить данную реализацию.

1. Если сопрограммы могут использовать общий стек данных, то можно сохранять и восстанавливать только указатель стека возвратов.

2. Можно передавать параметры при начале работы сопрограммы, а также при каждом возобновлении. Для этого потребуется специальное слово, пересылающее указанное число элементов стека вызывающей сопрограммы на стек вызываемой.

3. Предусмотрев поле связи для сцепления всех сопрограмм в список, можно получить состояние всей совокупности взаимодействующих сопрограмм.

4. Можно предусмотреть вызов асинхронного выхода «Окончание задачи». Идентификация этого выхода может быть как статической, так и динамической.

5. Часть действий можно выполнять в состоянии исполнения, а не компиляции. Среди этих действий могут быть следующие: создание сопрограммы без имени и отведение заказываемого участка памяти в динамически распределяемой памяти, инициализация этой области памяти, освобождение области памяти сопрограммы при ее завершении.

6. Если использовать косвенный вызов через переменную типа VECT , можно динамически определять возобновляемую сопрограмму.

Возможны, конечно, и любые другие изменения, диктуемые конкретной обстановкой.

3.9. **Запланированное перекрытие**

Адресное пространство форт-системы не так уж велико — 64 K байт. Это налагает известное ограничение на общее число форт-слов, одновременно присутствующих в словаре. Для снижения этого ограничения можно применить свернутый шитый код или использовать форт-систему, в которой в качестве основного вместо !6-разрядного значения выступает 32-разрядное. Можно заметно сократить требуемый объем оперативной памяти, если использовать векторное поле кода.

Опишем еще один простой способ снижения требований к памяти — *запланированное перекрытие*. Группа взаимосвязанных определений образует единый сегмент в двоичном коде, который может быть загружен на заранее заданное место в словаре из внешней памяти форт-системы. Разные сегменты, загружаемые на одно и то же место оперативной памяти, перекрывают друг друга, вот почему такая структура и названа запланированным перекрытием. Благодаря тому, что хранящийся во внешней памяти сегмент выражен в двоичном коде, а не в виде текста, его загрузка идет примерно в 50—!00 раз быстрее, чем обычная текстовая интерпретация.

В приводимой далее реализации каждый сегмент загружается на фиксированный адрес — значение указателя HERE на момент начала его компиляции. Определение сегмента включается в список FORTH .

После того как трансляция сегмента завершена, при его выгрузке во внешнюю память указатель вершины словаря понижается до прежнего значения с одновременным исключением всех слов данного сегмента из списка FORTH . При загрузке сегмента на соответствующий адрес указатель вершины словаря вновь поднимается, и определения сегмента снова включаются в список FORTH .

Для трансляции сегментов введем глобальную переменную (СЕГМ) , содержащую адрес начала очередного компилируемого сегмента, границы которого отмечаются словами СЕГМ-НАЧ и СЕГМ-КОН . Выгрузку и загрузку сегментов выполняют слова СЕГМ-ВЫГР и СЕГМ-ЗАГР .

```
VARIABLE (СЕГМ)
: СЕГМ-НАЧ  ( -> НАЧАЛО ТРАНСЛЯЦИИ СЕГМЕНТА)
   HERE DUP (СЕГМ) ! 0 ,       ( НОМЕР ЭКРАНА)
    HERE  0 C,  BL C,          ( ФИКТИВНОЕ ИМЯ)
   FORTH DEFINITIONS LATEST , ( ЗВЕНО СВЯЗИ)
      CURRENT @ !      ,      ( ТОЧКА ЗАГРУЗКИ)
      0 , 0 ,      ( ДЛИНА И ПОСЛЕДНЯЯ СТАТЬЯ)
;
: СЕГМ-КОН   ( -> КОНЕЦ ТРАНСЛЯЦИИ СЕГМЕНТА)
   (СЕГМ) @ >R     ( ТОЧКА ЗАГРУЗКИ СЕГМЕНТА)
   HERE R@ -  ( ДЛИНА) DUP  R@ 8 + !
   FORTH DEFINITIONS  LATEST  R> 10 + ! ;
: СЕГМ-ВЫГР ( N:НОМЕР ЭКРАНА->   ВЫГРУЗКА)
 DUP (СЕГМ) @ >R R@ !  ( НОМЕР)
 R@ SWAP ( A:АДРЕС СЕГМЕНТА,N) DUP R@ 8 + @
 1023 + 1024 / ( A,N,N,N1:ЧИСЛО ЭКРАНОВ)
  CR ." СЕГМЕНТ ВЫГРУЖЕН НА ЭКРАНЫ: " + SWAP DO
  I . ( AI:АДРЕС СЛЕДУЮЩЕЙ ПОРЦИИ) DUP I BUFFER
  1024 CMOVE UPDATE 1024 +LOOP DROP
   FLUSH R> (FORGET) ;
: СЕГМ-ЗАГР ( N:НОМЕР ЭКРАНА->  ЗАГРУЗКА)
  DUP BLOCK ( N,A0:АДРЕС ПЕРВОГО БЛОКА СЕГМЕНТА)
  2DUP @ - IF DROP CR . ." - БЛОК ДЛЯ ЗАГРУЗКИ?"
  ABORT THEN    DUP 6 + @ (FORGET) 2DUP
  HERE ( N,A0,N,A0,A1:НАЧАЛО) SWAP 8 + @ ( N,A0,N,A1,L)
  DUP ALLOT 1023 + 1024 / ( N,A0,N,A1,N1:ЧИСЛО ЭКР) ROT
  DUP ROT + SWAP ( ...N1+N,N) DO I BLOCK OVER 1024 CMOVE
   1024 +  LOOP 2DROP ( N) BLOCK 10 + @  CURRENT @ !  ;
```

Слово СЕГМ-НАЧ строит заголовок сегмента, который располагается в его начале и содержит начальный номер экрана во внешней памяти, фиктивную словарную статью, адрес начала сегмента в оперативной памяти, длину сегмента и адрес начала его последней словарной статьи. Номер экрана определяется программистом при выгрузке сегмента во внешнюю память, впоследствии именно с этого экрана начинается его загрузка. Слово СЕГМ-ЗАГР проверяет, содержат ли первые 2 байта

сегмента данный номер, и выдает сообщение об ошибке, если это не так.

Фиктивная словарная статья в заголовке сегмента аналогична полю параметров для слов, определенных через VOCABULARY , и служит для включения определений данного сегмента в список FORTH при загрузке сегмента. Она сцепляется с последней статьей списка FORTH на момент начала трансляции сегмента. При загрузке сегмента список FORTH устанавливается на последнее определение внутри данного сегмента. Таким образом, словарные статьи сегмента вновь включаются в словарь форт-системы. При изменениях состояний словаря, связанных с загрузкой и выгрузкой сегментов, в качестве вспомогательного используется слово (FORGET), которое корректным образом понижает вершину словаря, исключая из него все системные ссылки на исключаемые статьи. В качестве параметра это слово использует новый адрес вершины словаря (см. модель форт-системы в приложении 1).

3.10. Элементарная машинная графика

Машинная графика — чрезвычайно перспективная область для применения языка Форт. Поскольку для задач машинной графики постоянно ведется поиск все новых и новых способов их постановки и решения, то здесь как нигде требуется очень гибкий инструментальный язык, позволяющий быстро реализовывать и проверять на практике разные варианты решений. Принцип обратной оперативной связи между постановкой задачи и ее решением позволяет программисту добиваться существенно более высоких профессиональных результатов, чем при традиционных подходах.

Вместе с тем развитие машинной графики в значительной степени определяется наличием тех или иных аппаратных средств. Чрезвычайное разнообразие их конкретных характеристик делает невозможным создание какого-либо универсального языка, применимого для всех возможных случаев. Язык Форт позволяет программисту самому быстро создавать необходимые инструментальные средства, применяя универсальные приемы с учетом особенностей конкретной задачи.

Рассмотрим две очень разные по сложности задачи построения графиков функций. В первой используется

алфавитно-цифровой терминал, во второй — графопостроитель с широким набором возможностей.

В случае использования алфавитно-цифрового терминала задача состоит в том, чтобы отобразить в нем график функции, заданной вектором своих значений. Введем понятие вектора, элементы которого нумеруются от нуля, по аналогии со словом QUAN (см. п. 3.4). Для получения адреса элемента вектора и для присваивания ему нового значения можно использовать те же слова AT и TO .

```
: ?+ ( N->N) DUP 0< ABORT" ОТРИЦАТЕЛЬНОЕ ЗНАЧЕНИЕ" ;
: В-АДР  ( I:ИНДЕКС,PFA->A[I]:АДРЕС)  2DUP @ U<
    IF SWAP 1+ 2* + EXIT THEN   SWAP CR
  . ." - НЕДОПУСТИМЫЙ ИНДЕКС ДЛЯ ВЕКТОРА "
   BODY> BODY> BODY>  >NAME  ID.  ABORT ;
: В-АДРО  DOES> ( I:ИНДЕКС,PFA->A[I]:АДРЕС)  В-АДР ;
: В-ПРИСВ DOES> ( N:ЗНАЧ,I:ИНДЕКС,2CFA->) 2+ В-АДР ! ;
: VQUAN  ( N:ВЕРХН.ИНДЕКС->) ?+ CREATE В-ПРИСВ HERE 2- @ ,
   В-АДРО HERE 4 - @ ,  1+ DUP ,  2* HERE SWAP DUP ALLOT
   ERASE  DOES> ( I:ИНДЕКС,1CFA->N[I]:ЗНАЧ)  4 + В-АДР @ ;
: В-РАЗМ  ( ->N:ЧИСЛО ЭЛЕМЕНТОВ)
   ' 6 + @  [COMPILE] LITERAL ;  IMMEDIATE
```

Невысокая разрешающая способность алфавитно-цифрового терминалах точки зрения машинной графики вполне допускает использование стандартных целых чисел для представления значений вещественных функций. Более того, такие дискретные вычислительные приемы оказываются значительно более удобными для задач машинной графики, чем работа с числами в формате плавающей точки. Например, тригонометрические функции можно вычислять следующим образом, задавая их аргумент в градусной мере:

```
( 18.06.86 SIN  COS)   ( ТАБЛИЦА СИНУСОВ ОТ 0 ДО 90 ГРАД.) HERE
00000 , 0175 , 0349 , 0523 , 0698 , 0872 , 1045 , 1219 , 1392 ,
 1564 , 1736 , 1908 , 2079 , 2250 , 2419 , 2588 , 2756 , 2924 ,
 3090 , 3256 , 3420 , 3584 , 3746 , 3907 , 4067 , 4226 , 4384 ,
 4540 , 4695 , 4848 , 5000 , 5150 , 5249 , 5446 , 5592 , 5736 ,
 5878 , 6018 , 6157 , 6293 , 6428 , 6561 , 6691 , 6820 , 6947 ,
 7071 , 7193 , 7314 , 7431 , 7547 , 7660 , 7771 , 7880 , 7986 ,
 8090 , 8192 , 8290 , 8387 , 8480 , 8572 , 8860 , 8746 , 8829 ,
 8910 , 8988 , 9063 , 9135 , 9205 , 9772 , 9336 , 9397 , 9455 ,
 9511 , 9563 , 9613 , 9659 , 9703 , 9744 , 9781 , 9816 , 9848 ,
 9877 , 9903 , 9925 , 9945 , 9962 , 9976 , 9986 , 9994 , 9998 ,
10000 ,    : SIN180 ( N->SIN N, 0<=N<=180)
DUP 90 > IF 180 SWAP - THEN 2* [ DUP ] LITERAL + @ ;      DROP
: SIN  ( N->SIN N)  360 MOD DUP 0< IF 360 + THEN
      DUP 180 > IF 180 - SIN180 NEGATE ELSE SIN180 THEN ;
: COS  ( N->COS N)   90 SWAP - SIN ;
```

Аналогично и другие элементарные функции можно с успехом вычислять по классическим итерационным схемам:

```
( DSQRT  SQRT  ИЗВЛЕЧЕНИЕ КОРНЯ ПО СХЕМЕ НЬЮТОНА)
: DSQRT ( D1->D2)  2DUP
                 D0< ABORT" ОТРИЦАТЕЛЬНЫЙ АРГУМЕНТ"
  2DUP 2. D< IF EXIT THEN
( ДАЛЕЕ ПО СХЕМЕ: X[0]=X/2; X[I+1]=X[I]/2+X/X[I]/2)
  2DUP  2. D/ SWAP
  BEGIN 2OVER 2OVER D/ 2OVER D+ 2. D/ 2SWAP 2OVER
        D- DABS 2. D< UNTIL
  2SWAP 2DROP ;
: SQRT  ( N1->N2)  S>D  DSQRT  DROP ;
```

Некоторое неудобство при таком представлении функций доставляет необходимость помнить о масштабном множителе, но это с лихвой возмещается простотой приемов их вычисления.

Используя описанные вспомогательные средства, можно задавать векторы со значениями интересующих нас функций. Работая с простыми значениями, будем пользоваться их описанием через слово QUAN. Пусть наша функция в обычной записи имеет вид: $y = \sin 3x + \sin 120x$. Зададим ее определение через введенные инструментальные определения, выбрав 100 в качестве масштабного множителя:

```
70 VQUAN U
: U!  ( -> УСТАНОВКА ВЕКТОРА U)  B-РАЗМ U  0  DO
     3 I * SIN  120 I * SIN + 100 / I TO U  LOOP ;
 U!
```

При желании можно проверить правильность установки значений, распечатав их как числа:

```
: B-?  ( ->  РАСПЕЧАТКА ВЕКТОРА)  >IN @ >R
    [COMPILE] B-РАЗМ  R> >IN ! ' SWAP 0 DO
      I OVER EXECUTE  8 .R  LOOP  DROP  ;
B-? U
```

В результате будет напечатан следующий текст:

```
   0      93     -78      15     109     -62      30     124
 -47      45     138     -34      58     151     -21      70
 162     -10      80     172       0      89     179       4
  95     185       9      98     188      11     100     188
  10      98     186       7      95     181       2      89
```

```
177    -4    80   166   -14    70   155   -25
 58   143   -38    45   129   -52    30   114
-67    15    99   -63     0    63   -99   -15
 67  -114   -30    52  -129   -45 OK
```

Для построения графика определим список слов PLOT (график) и в нем ряд вспомогательных переменных:

```
VOCABULARY PLOT   PLOT DEFINITIONS
QUAN RHO   4 TO RHO    ( ШИРИНА СТОЛБЦА ДЛЯ Y-КООРД.)
QUAN MU0  16 TO MU0    ( МАКСИМАЛЬНАЯ ВЫСОТА ОКНА)
QUAN NU0  64 RHO - TO NU0 ( МАКСИМАЛЬНАЯ ШИРИНА ОКНА)
QUAN WSTA              ( АДРЕС НАЧАЛА ОКНА)
QUAN MU                ( ТЕКУЩАЯ ВЫСОТА ОКНА)
QUAN NU                ( ТЕКУЩАЯ ШИРИНА ОКНА)
QUAN WX                ( X-КООРДИНАТА ОКНА)
QUAN WY                ( Y-КООРДИНАТА ОКНА)
QUAN KAPPA             ( ЧИСЛО ЗНАЧЕНИЙ СИГНАЛА)
QUAN LAMBDA            ( ДИАПАЗОН ЗНАЧЕНИЙ СИГНАЛА)
QUAN MA                ( МАКСИМАЛЬНОЕ ЗНАЧЕНИЕ СИГН.)
QUAN MINO              ( МИНИМАЛЬНОЕ ЗНАЧЕНИЕ В ОКНЕ)
VECT VAL               ( ВЕКТОР СИГНАЛА)
```

Пусть прямоугольный экран терминала, на котором строится график, содержит MU0 строк по NU0 литер каждая. Первые RHO позиций в каждой строке занимает надпечатка ее координаты по оси ординат. Общий размер графика определяется значениями LAMBDA (диапазон значений функций) и KAPPA (число значений) . Этот прямоугольник может как умещаться целиком на экране терминала, так и заметно превышать его. Определим понятие «окна», которое перемещается по графику и показывает на экране терминала соответствующую его часть. Переменные WX и WY задают координаты левого верхнего угла окна относительно левого верхнего угла графика. Определим слова, выполняющие инициализацию и перемещение окна по полю графика:

```
FORTH DEFINITIONS PLOT
: НАЧАТЬ ( -> ИНИЦИАЛИЗАЦИЯ)    PLOT
  HERE NU0 MU0 * ALLOT TO WSTA ;
PLOT DEFINITIONS
: ВВЕРХ  ( N->  СДВИНУТЬ ОКНО НА N ПОЗИЦИЙ ВВЕРХ)
   ?+ WY SWAP - 0 MAX TO WY ;
: ВЛЕВО ( N->  СДВИНУТЬ ОКНО НА N ПОЗИЦИЙ ВЛЕВО)
   ?+ WX SWAP - 0 MAX TO WX ;
: ВНИЗ  ( N->  СДВИНУТЬ ОКНО НА N ПОЗИЦИЙ ВНИЗ)
   ?+ WY + MU + LAMBDA MIN MU - TO WY ;
: ВПРАВО ( N-> СДВИНУТЬ ОКНО НА N ПОЗИЦИЙ ВПРАВО)
   ?+ WX + NU + KAPPA  MIN NU - TO WX ;
```

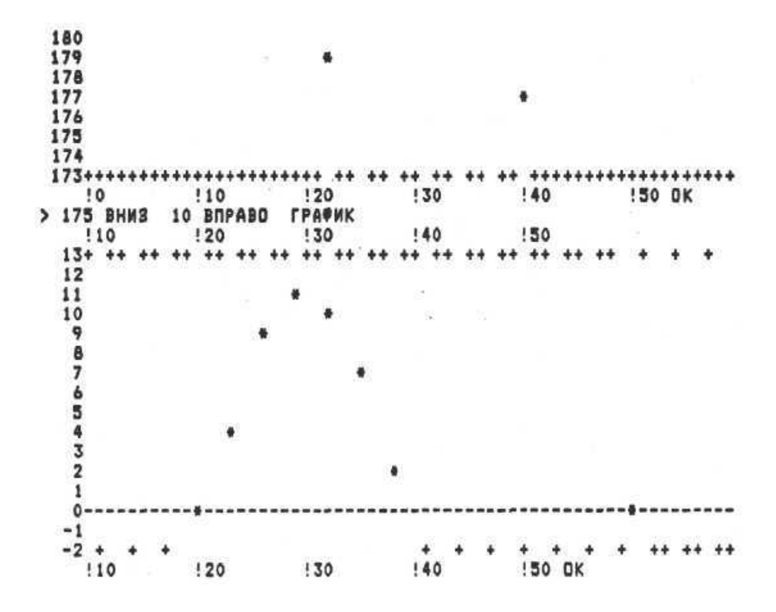
Определим также слова для вычисления параметров графика и распечатки окна по его текущим координатам. Вспомогательное слово MU по значению функции

определяет, где находится соответствующая точка графика: ниже окна, внутри или выше него. Если точка попадает в окно, будем отмечать ее положение знаком * (звездочка), а если точка находится выше или ниже окна, то соответственно в самой верхней или самой нижней строке окна поставим знак + (плюс), чтобы видеть направление, в котором следует искать данную точку на графике. Ось абсцисс, если она попадает в окно, будем отмечать строкой из знаков «минус». Надпечатку координат по оси абсцисс будем выполнять дважды — в верхней и нижней строках терминала.

```
: -MU ( N:ЗНАЧЕНИЕ->-1:НИЖЕ ОКНА/
                   M:СМЕЩЕНИЕ ОТ ВЕРХА ОКНА ВНИЗ,0/
                   +1:ВЫШЕ ОКНА )
   MINO - DUP 0< IF DROP -1 EXIT THEN
         MU 1- SWAP - DUP 0< IF DROP 1 EXIT THEN 0 ;
: ФУНКЦИЯ ( -> СЛЕДУЮЩЕЕ СЛОВО: ИМЯ ВЕКТОРА ЗНАЧЕНИЙ)
  >IN @ >R [COMPILE] В-РАЗМ  R> >IN !
  DUP TO KAPPA NUO MIN TO NU   ' TO VAL
  ( ВЫЧИСЛЕНИЕ MA И LAMBDA )
  0 VAL DUP   ( X[0],X[0] )
  KAPPA 0 DO I VAL ( MIN[I-1],MAX[I-1],X[I] )
  DUP >R MAX SWAP R> MIN ( MAX[I],MIN[I]) SWAP LOOP
  DUP TO MA  SWAP - 1+ DUP TO LAMBDA  MUO MIN TO MU
  0 TO WX  0 TO WY ;
: ОСЬ-X ( НАДПЕЧАТКА ОСИ X) CR RHO SPACES WX NU 1- RHO - WX
  BEGIN DUP 10 MOD DUP IF 10 SWAP - THEN DUP >R + 2DUP >=
  WHILE R> SPACES C" ! EMIT DUP 0 <# #S #> DUP >R TYPE
        1+ R> + REPEAT  RDROP 2DROP ;
: ГРАФИК ( -> РИСОВАТЬ ГРАФИК ПО СИГНАЛУ И КООРДИНАТАМ ОКНА)
  MA WY - MU 1- - TO MINO ( УСТАНОВИТЬ MINO)
  WSTA MU NU * BLANK ( ЗАЧИСТИТЬ ОКНО)
  0 -MU 0= IF NU * WSTA + NU C" - FILL THEN ( ПРОВЕСТИ ОСЬ X)
  ( ЦИКЛ ПО ОСИ X ДЛЯ ЗАПОЛНЕНИЯ ОКНА)  WX NU + WX DO
    I VAL ( X[I])  -MU DUP IF ( ВНЕ ОКНА) 1+ IF ( ВЫШЕ ОКНА)
      C" + 0 ELSE ( НИЖЕ ОКНА) C" + MU 1- THEN
                         ELSE DROP C" * SWAP THEN
    NU * I WX - + WSTA + C! LOOP
ОСЬ-X  MU 0 DO CR  MA WY - I - RHO .R  I NU * WSTA +
                NU TYPE LOOP  ОСЬ-X ;
```

Следующий протокол показывает работу введенных определений по вычерчиванию графика интересующей нас функции:

```
> НАЧАТЬ  ФУНКЦИЯ U  ГРАФИК
   !0        !10       !20        !30       !40        !50
 188                             *  *
 187
 186                                   *
 185                          *
 184
 183
 182
 181                                      *
```



Вычерчивание графиков на графопостроителе чрезвычайно осложняется обилием детален чертежа, которые надо учитывать. При этом собственно вычисление значений функции составляет незначительную часть в общем объеме работы. Многие элементы чертежа имеют, как правило, какое-нибудь стандартное значение, принимаемое по умолчанию, и программист может их только переопределять (например, заголовок, штамп, оформление полей, выбор шрифтов и цвета надписей и т. д.). С каждым из этих крупных элементов можно связать список слов, обрабатывающих те или иные его компоненты, и для установки нужного значения исполнять соответствующее слово в контексте данного элемента. Сходные действия (например, задание шрифта или цвета) в разных контекстах могут обозначаться одними и теми же словами, что создает большие удобства для программиста. Вместе с тем и синтаксис для задания действий графопостроителя можно сделать максимально простым и удобным для пользователей-непрограммистов. Например, определение элементов чертежа можно задавать так:

```
ГРАФИК  XY  X  ОТ  0  ДО  4  ШАГ  1
Y  ЛОГАРИФМ  ОТ  1  ДО  100
```

```
X ШТРИХ РАЗМЕР 0.25 РАМКА
ЗАГОЛОВОК" ЧЕРТЕЖ 1" РИСУЙ
```

В результате будет нарисована прямоугольная рамка с заголовком «ЧЕРТЕЖ 1» надпечаткой оси абсцисс в виде штрихов, пересекающих рамку, и логарифмической разметкой оси ординат. Огромное число других значений, необходимых для построения чертежа, принимается по умолчанию. Далее можно начертить собственно график функции, используя, например, ее задание через вектор значений:

```
ФУНКЦИЯ U  МАСШТАБ 100  ЦВЕТ КРАСНЫЙ  РИСУЙ
```

Аналогично наносим на чертеж дополнительные элементы:

```
ПОДЗАГОЛОВОК ВПРАВО
ТЕКСТ" Y=SIN 3X + SIN 120X"  РИСУЙ
```

и т. д. Указанные действия можно не только непосредственно исполнять, но и компилировать, задавая впоследствии их исполнение через одно слово — имя скомпилированной программы.

3.11. Реализация встроенного ассемблера

Приводимый ниже текст является законченной реализацией встроенного структурного ассемблера для форт-систем на базе микропроцессора К580. В оттранслированном виде он занимает около 1300 байт и является типичным для 8-разрядных микропроцессоров. Для 16- или 32-разрядных процессоров в силу их большей сложности объем текста больше. Но даже в этом случае объем исходного текста и скомпилированного кода существенно меньше, чем для традиционных ассемблеров.

```
(  FORTH-83 АССЕМБЛЕР ДЛЯ К580 ТЕМЯШКИНО  1985 )
    VOCABULARY ASSEMBLER
    ASSEMBLER DEFINITIONS
       (  МАШИННЫЕ КОМАНДЫ  )
DECIMAL         : 8* 2* 2* 2* ;
4 CONSTANT H  5 CONSTANT L 7 CONSTANT A  6 CONSTANT PSW
2 CONSTANT D  3 CONSTANT E 0 CONSTANT B  1 CONSTANT C
6 CONSTANT M  6 CONSTANT SP
 : 1MI  ( ---> ) CREATE C, DOES> C@ C, ;
 : 2MI  ( ---> ) CREATE C, DOES> C@ + C, ;
 : 3MI  ( ---> ) CREATE C, DOES> C@ SWAP 8* + C, ;
 : 4MI  ( ---> ) CREATE C, DOES> C@ C, C,   ;
 : 5MI  ( ---> ) CREATE C, DOES> C@ C,  ,   ;
```

```
HEX
00 1MI NOP   76 1MI HLT   F3 1MI DI    FB 1MI EI
07 1MI RLC   0F 1MI RRC   17 1MI RAL   1F 1MI RAR
E9 1MI PCHL  F9 1MI SPHL  E3 1MI XTHL  EB 1MI XCHG
27 1MI DAA   2F 1MI CMA   37 1MI STC   3F 1MI CMC
80 2MI ADD   88 2MI ADC   90 2MI SUB   98 2MI SBB
A0 2MI ANA   A8 2MI XRA   B0 2MI ORA   B8 2MI CMP
09 3MI DAD   C1 3MI POP   C5 3MI PUSH  02 3MI STAX
0A 3MI LDAX  04 3MI INR   05 3MI DCR   03 3MI INX
0B 3MI DCX   C7 3MI RST   D3 4MI OUT   DB 4MI IN
C6 4MI ADI   CE 4MI ACI   D6 4MI SUI   DE 4MI SBI
E6 4MI ANI   EE 4MI XRI   F6 4MI ORI   FE 4MI CPI
22 5MI SHLD  2A 5MI LHLD  32 5MI STA   3A 5MI LDA
C4 5MI CNZ   CC 5MI CZ    D4 5MI CNC   DC 5MI CC
E4 5MI CPO   EC 5MI CPE   F4 5MI CP    FC 5MI CM
CD 5MI CALL  C9 1MI RET   C3 5MI JMP
C0 1MI RNZ   C8 1MI RZ    D0 1MI RNC   D8 1MI RC
E0 1MI RPO   E8 1MI RPE   F0 1MI RP    F8 1MI RM
C2 5MI JNZ   CA 5MI JZ    D2 5MI JNC   DA 5MI JC
E2 5MI JPO   EA 5MI JPE   F2 5MI JP    FA 5MI JM
: MOV SWAP 8* 40 + + C, ;
: MVI SWAP 8* 6  + C, C, ;
: LXI SWAP 8* 1 + C, , ;
  (  КОДЫ УСЛОВИЯ ДЛЯ СТРУКТУР УПРАВЛЕНИЯ  )
C2 CONSTANT 0=    D2 CONSTANT CS
E2 CONSTANT PE   F2 CONSTANT 0<
  (  СТРУКТУРЫ УПРАВЛЕНИЯ  )
DECIMAL
: NOT    ( N:КОД УСЛОВИЯ ---> N1) 8 + ;
: THEN   ( АДР,2 ---> ) 2 ?PAIRS HERE SWAP ! ;
: IF     ( КОД ---> АДР,2 ) C, HERE 0 , 2 ;
: ELSE   ( АДР,2 ---> АДР1,2) 2 ?PAIRS C3 IF ROT SWAP THEN  2 ;
: BEGIN  ( ---> АДР,1 )  HERE 1 ;
: UNTIL  ( АДР,1,КОД --->) SWAP 1 ?PAIRS C, , ;
: AGAIN  ( АДР,1 --->) 1 ?PAIRS C3 C, , ;
: WHILE  ( АДР,1,КОД ---> АДР,1,АДР1,4)  IF 2 + ;
: REPEAT ( АДР,1,АДР1,4 --->) >R >R AGAIN R> R> 2- THEN ;
  (  РАБОТА С МЕТКАМИ  )
 10 CONSTANT LBLMAX ( МАКСИМАЛЬНОЕ ЧИСЛО ЛОКАЛЬНЫХ МЕТОК)
    VARIABLE LTABLE   LBLMAX 1+ 2*  ALLOT
 10 CONSTANT FRMAX ( МАКСИМАЛЬНОЕ ЧИСЛО ССЫЛОК ВПЕРЕД)
    VARIABLE FRTABLE  FRMAX  2* 2*  ALLOT
 : FRCHK    ( ---> ПРОВЕРКА НЕРАЗРЕШЕННЫХ ССЫЛОК ВПЕРЕД )
            FRMAX 0 DO I 2* 2* FRTABLE + @
                       ABORT" НЕРАЗРЕШЕННАЯ ССЫЛКА ВПЕРЕД"
                   LOOP ;
 : FRCLR    ( ---> ИНИЦИАЛИЗАЦИЯ ТАБЛИЦЫ ССЫЛОК ВПЕРЕД )
            FRTABLE FRMAX 2* 2* ERASE
            LTABLE LBLMAX 1+ 2* ERASE ;

 : FRRES    ( N:МЕТКА --->     РАЗРЕШЕНИЕ ССЫЛОК ВПЕРЕД )
            FRMAX 0 DO  I 2* 2* FRTABLE + 2DUP @ =
                       IF HERE OVER 2+ @ +! 0!
                       ELSE DROP THEN
                   LOOP ;
 : FRADD    ( N:МЕТКА ---> ДОБАВЛЕНИЕ ССЫЛКИ ВПЕРЕД В ТАБЛИЦУ )
            FRMAX 1+ 0
            DO FRMAX I = ABORT" СЛИШКОМ МНОГО ССЫЛОК ВПЕРЕД"
               I 2* 2* FRTABLE + DUP @ 0=
               IF 2DUP ! HERE 1+ SWAP 2+ ! LEAVE ELSE DROP THEN
            LOOP ;
```

```
   (   ОПРЕДЕЛЯЮЩИЕ ВХОЖДЕНИЯ МЕТОК    )
: !LT      CREATE , DOES> @  FRRES HERE SWAP 2* LTABLE + ! ;
           LOOP ;
   (   ОПРЕДЕЛЯЮЩИЕ ВХОЖДЕНИЯ МЕТОК    )
: !LT      CREATE , DOES> @  FRRES HERE SWAP 2* LTABLE + ! ;
   (   ИСПОЛЬЗУЮЩИЕ ВХОЖДЕНИЯ МЕТОК    )
: @LT      CREATE , DOES> @ DUP 2*
           LTABLE + @ SWAP OVER 0=
           IF FRADD THEN DROP ;
1 !LT 1#:   2 !LT 2#:  3 !LT 3#:  4 !LT 4#:  5 !LT  5#:
6 !LT 6#:   7 !LT 7#:  8 !LT 8#:  9 !LT 9#: 10 !LT 10#:
1 @LT 1#    2 @LT 2#   3 @LT 3#   4 @LT 4#   5 @LT  5#
6 @LT 6#    7 @LT 7#   8 @LT 8#   9 @LT 9#  10 @LT 10#
     (   ПЕРЕКЛЮЧЕНИЕ В АССЕМБЛЕР    )
             FORTH DEFINITIONS
: BEG-ASM [ ASSEMBLER ] FRCLR [ FORTH ] ASSEMBLER
           !CSP ;
: END-ASM [ ASSEMBLER ] FRCHK [ FORTH ] ?CSP
```

В приведенном тексте можно выделить четыре группы определений: машинные команды, структуры управления, метки и стандартные слова для создания определений в машинном коде. Помимо них данный текст определяет ряд вспомогательных слов.

Машинные команды на языке ассемблера записываются в обратной польской форме, принятой для языка Форт: <операнды> <операция> , где <операнды> — слова, вычисляющие на стеке размещения операндов данной машинной команды, а <операция> — ее мнемоника. Для обозначения регистров микропроцессора K580 зарезервированы слова A B C D E H L, для задания регистровых пар используются слова B D H SP PSW , для косвенной адресации через регистровую пару HL — слово M.

При исполнении слов, обозначающих регистры, регистровые пары и косвенную адресацию, на стеке оставляются значения, соответствующие принятым в системе команд K580 обозначениям для регистров и регистровых пар:

```
A  B  C  D  E  H  L  M  SP  PSW
7  0  1  2  3  4  5  6  6   6
```

При исполнении мнемоники машинной команды на вершину словаря компилируется соответствующая машинная инструкция. Размещения операндов при этом снимаются со стека. Порядок вычисления операндов машинной команды соответствуют порядку написания операндов на обычном языке ассемблера.

Ниже приведены для сравнения записи машинных команд на обычном языке ассемблера и на языке ассемблера данной форт-системы:

```
HLT                 HLT
DAD  SP             SP DAD
PUSH PSW            PSW PUSH
CMP  C              C CMP
MOV  A,M            A M MVI
MVI  B,2            B 2 MVI
LXI  D,120+8        D 120 8 + LXI
CALL 5002H          HEX 5002 CALL DECIMAL
```

Для определения мнемоник используются определяющие слова, соответствующие форматам машинных команд (1MI, ..., 5MI). Все они имеют одинаковую создающую часть, которая компилирует в поле параметров статьи для мнемоники однобайтный код маски для кода данной команды, который снимается со стека. Исполняющая часть определений, используя маску и размещения операндов, которые она снимает со стека, компилирует двоичный код, соответствующий данной команде. Так, например, через елово 1MI определяются машинные команды, не имеющие операндов. При исполнении мнемоники такой команды на вершину словаря компилируется однобайтный код операции. Слово 3MI определяет мнемоники команд, имеющих один операнд — номер регистра. Этот номер занимает разряды со 2 по 4 в однобайтном коде команды. Поэтому исполняющая часть сдвигает свой операнд — номер регистра — влево на 3 разряда исполнением слова 8* и добавляет к нему маску команды. Получившийся однобайтный код компилируется на вершину словаря. Три команды — MOV, MVI и LXI — не подходят под описанные общие форматы, поэтому они определяются непосредственно через двоеточие.

Структурные операторы встроенного ассемблера позволяют программировать разветвления и циклы без явно заданных команд перехода на метку. Их синтаксис аналогичен синтаксису соответствующих операторов языка Форт.

Условный оператор в полной или сокращенной форме записывается следующим образом:

```
<КОД-УСЛОВИЯ> IF <ЧАСТЬ-ТО> ELSE <ЧАСТЬ-ИНАЧЕ> THEN
<КОД-УСЛОВИЯ> IF <ЧАСТЬ-ТО> THEN
```

Часть «то» выполняется, только если в разрядах PSW установлен заданный «код условия», в противном случае выполняется часть «иначе», если она есть.

Циклы могут быть записаны в одной из трех форм:

```
BEGIN <ТЕЛО-ЦИКЛА> AGAIN
BEGIN <ТЕЛО-ЦИКЛА> <КОД-УСЛОВИЯ> UNTIL
BEGIN <ТЕЛО-1> <КОД-УСЛОВИЯ> WHILE <ТЕЛО-2> REPEAT
```

Цикл BEGIN-AGAIN является бесконечным, BEGIN-UNTIL прекращает выполнение при к указанном коде условия, в цикле BEGIN-WHILE-REPEAT, наоборот, указанный код условия задает продолжение цикла.

Для указания кода условия в ассемблере используются следующие слова: 0= — установлен флаг Z, 0< — флаг S, CS — флаг C, PE — флаг P. Для инвертирования кода условия используется слово NOT.

Структурные операторы компилируют на своем месте команды безусловных и условных переходов с указанным условием.

Для работы с явно задаваемыми метками в ассемблер включены слова вида 1#: 2#: ... 10#: и 1# #2# ... 10#. Слова первой группы задают определение метки, слова второй — использование. Метки имеют строго локальный характер, т. е. эти слова можно использовать только внутри определений через CODE и между словами ; CODE и END-CODE . Наряду с метками допускается использование ассемблерных структур управления.

Локальные метки рекомендуется использовать только в трехбайтных командах типа JMP, CALL, LXI . Уже определенные локальные метки могут участвовать в арифметических и стековых операциях; использующие вхождения меток, определяющие вхождения которых вводятся позднее (ссылки вперед), могут участвовать только в операциях, сводящихся к добавлению к ним числа со знаком.

Для реализации работы с метками ассемблер имеет таблицу меток LTABLE и таблицу ссылок вперед FRTABLE . Их размер определяет максимальное число разных меток и ссылок вперед в пределах одного определения в машинном коде (от слова CODE или ; CODE до слова END-CODE). Для каждой возможной метки соответствующий элемент таблицы LTABLE содержит нуль, если метка еще не определена, или адрес, который является ее значением, если метка уже определена. Входом в эту таблицу служит номер метки. В таблице FRTABLE для каждой ссылки вперед хранятся два

значения: номер метки и адрес в словаре, по которому нужно вписать ее значение. По этому адресу скомпилировано значение смещения, которое нужно добавить как число со знаком к значению данной метки.

Слова для использующих вхождений меток определяются через @LT, например 1 @LT 1#. Исполнение определенного таким образом слова 1 #, обозначающего метку номер 1, состоит в обращении к таблице LTABLE по индексу 1. Если там стоит нуль, т. е. метка номер 1 еще не определена, то этот нуль выдается на стек в качестве смещения от значения метки, которое определится позднее. Одновременно в таблицу FRTABLE заносится запись о данной ссылке вперед. В качестве адреса, куда нужно будет впоследствии вписать значение метки, берется HERE 1+. Здесь используется тот факт, что в микропроцессоре К580 операнд-адрес занимает 2 байта, следующие за однобайтным кодом операции.

Слова для определения меток определяются через слово !LT , например 1 !LT 1 # : . Исполнение определенного таким образом слова 1#: определяет метку номер 1 как текущий адрес вершины словаря HERE . Его действие состоит в том, что выполняется просмотр таблицы FRTABLE с разрешением всех накопившихся ссылок на данную метку, после чего запись о данной метке заносится в таблицу LTABLE .

Последнюю группу определений составляют слова для доступа к встроенному ассемблеру при компиляции машинного кода. Помимо стандартных форт-слов CODE, ; CODE и END-CODE здесь определяются слова LABEL и NEXT; . Первое используется для создания именованных подпрограмм, к которым можно обращаться из машинного кода. Второе является сокращением для часто встречающегося окончания ассемблерных определений: NEXT JMP END-CODE . В качестве примера приведем определение слова > < , которое меняет местами байты в переданном на стеке двухбайтном значении:

```
CODE >< ( N1 ---> N2)   H POP
   A L MOV  L H MOV  H A MOV  H PUSH  NEXT;
```

Приведенная реализация встроенного ассемблера может быть усовершенствована по нескольким направлениям. Прежде всего, увеличив константы LBLMAX и

FRMAX и дабавив новые определения для слов, обозначающих метки, можно увеличить количество разных меток и ссылок вперед, которые разрешается использовать в ассемблерном определении. Далее можно ввести контроль правильности операндов, включив соответствующие проверки в исполняющую часть определяющих слов для мнемоник команд. Для проверки числа элементов на стеке обычно используется глобальная ячейка CSP и слово !CSP, которое засылает в нее текущий адрес вершины стека. Интересным расширением является возможность введения макрокоманд. Макрокоманды без локальных меток можно определять обычным образом через двоеточие:

```
INRN  ( R:РЕГИСТР,N:ЧИСЛО РАЗ ---))
[ ASSEMBLER ]  0 DO DUP INR LOOP  DROP  ;
```

При исполнении текста A 3 INRN будут скомпилированы три команды A INR.

Если же макрорасширение должно порождать обращения к адресам через локальные метки, то потребуется более основательное расширение ассемблера.

Приложение 1.

Модель форт-системы

Приводимый ниже текст представляет собой ядро системы ФОРТ-ЕС (см. приложение 2), из которого исключены запускающая часть и реализации слов нижнего уровня для обмена с терминалом и внешней памятью). Общий объем ядра — 8 К байт (свыше 200 слов). Текст состоит из двух частей — списка слов с их краткими спецификациями и экранов с определениями на встроенном языке ассемблера и на языке Форт.

Помимо слов, которым соответствуют статьи в словаре, внутри ассемблерных определений используются метки и адреса. В спецификациях эти объекты отмечены буквами M и A. Они определяются с помощью слов M: и A: соответственно. Метки используются в машинных командах, а адреса порождают двухбайтное значение, содержащее данный адрес. Адресные операнды FIRST и SECOND обозначают соответственно первый и второй элементы стека. Макрокоманды PUSH, ,POP. и PULL можно рассматривать как команды с одним

регистровым операндом. Операция PUSH, помещает на стек значение из регистра, POP, снимает верхнее" значение со стека, засылая его в регистр, и PULL, копирует верхнее значение стека в регистр. Кроме того, в ассемблерных определениях используются локальные метки [11, с.191], обозначаемые целыми числами и словами =F (для ссылки вперед) и =B (для ссылки назад). Определяется локальная метка через слово = H, которое полагает ее равной текущему значению счетчика адреса. Регистры общего назначения обозначаются специальными словами и имеют следующий смысл:

RW1 — рабочий регистр, старший в паре;

RW2 — рабочий регистр, младший в паре;

RI — указатель адресного интерпретатора;

RRET — абсолютный адрес вершины стека возвратов;

RSTACK — абсолютный адрес вершины стека данных;

RD — форт-адрес текущей вершины словаря;

RFORTH — абсолютный адрес начала словаря, соответствующий нулевому форт-адресу;

RNEXT — адрес точки NEXT адресного интерпретатора (тот же адрес, что и в RFORTH);

RTWO — константа 2;

RMASK — константа 65535.

Предполагается также, что в регистре 13 находится адрес области сохранения и регистры 0, 1, 14, 15 свободно используются внутри определений как рабочие.

Тексты определений представлены в виде распечаток стандартных форт-текстов и занимают экраны с номерами от 1 до 47. В последнем столбце спецификации для каждого слова указан номер экрана, на котором оно определено. Главным словом модели является слово ФОРТ-СИСТЕМА.

В списке спецификаций слова расположены по возрастанию в кодировке ДКОИ. Они могут иметь следующие отметки:

А — адрес;

М — метка;

Н — слово немедленного исполнения;

К — требуется режим компиляции;

П — переменная, размещенная в пользовательской области;

С — системная переменная, размещенная в словаре;

Э — требуется режим обработки экрана;

\+ — слово из дополнения к стандарту "Форт-83";

* — нестандартное слово.

Для каждого слова указываются значения, которые оно снимает со стека (слева от знака →), и результат, который оно оставляет на стеке (справа от знака →). Если перечисляется несколько значении, то верхнее (на вершине стека) находится справа.

Для задания параметров и результатов, передаваемых через стек данных, используются следующие обозначения:

\+ N — неотрицательное целое со знаком;

A — двухбайтный форт-адрес;

C — однобайтное значение (старший байт, как правило, нулевой);

CFA — двухбайтный адрес поля кода словарной статьи;

D — четырехбайтное целое со знаком;

F — булевское значение (0 — ЛОЖЬ, не 0 — ИСТИНА);

FF — булевское значение ЛОЖЬ (0);

L — абсолютный машинный адрес (четырехбайтный);

LFA — двухбайтный адрес поля словарной статьи;

N — двухбайтное целое со знаком;

NFA — двухбайтный адрес поля имени словарной статьи;

PFA — двухбайтный адрес поля параметров словарной статьи;

T — двухбайтный адрес строки со счетчиком;

TF — булевское значение ИСТИНА (не 0, обычно -1);

U — двухбайтное целое без знака;

UD — четырехбайтное целое без знака;

W — двухбайтное целое со знаком или без него (N или U);

WD — четырехбайтное целое со знаком или без него (D или UD).

Для некоторых слов указаны две группы результатов, отделяемые друг от друга косой чертой (/). Они резличаются по булевскому значению на вершине стека: не 0 (ИСТИНА) означает успех, 0 (ЛОЖЬ) — неудачу.

```
           *H ->  (ПУСТОЕ СЛОВО) - ЗАКОНЧИТЬ ИНТЕРПРЕТАЦИЮ  40
                                          ВХОДНОГО ПОТОКА
[           H ->       ПЕРЕКЛЮЧИТЬ ТЕКСТОВЫЙ ИНТЕРПРЕТАТОР  22
                                       В РЕЖИМ ИСПОЛНЕНИЯ
[']        HK ->   /КОМПИЛЯЦИЯ/       СКОМПИЛИРОВАТЬ CFA  41
              ->CFA /ИСПОЛНЕНИЕ/     СЛЕДУЮЩЕГО СЛОВА КАК
                                         ЧИСЛОВОЙ ЛИТЕРАЛ
[COMPILE] HK ->            СКОМПИЛИРОВАТЬ СЛЕДУЮЩЕЕ СЛОВО  41
                    НЕЗАВИСИМО ОТ ЕГО ПРИЗНАКА "IMMEDIATE"
.             N->  НАПЕЧАТАТЬ N НА ТЕРМИНАЛЕ И ДАТЬ ПРОБЕЛ  38
.(          H ->           НАПЕЧАТАТЬ СЛЕДУЮЩИЕ ЛИТЕРЫ ДО  28
                         ЗАКРЫВАЮЩЕЙ СКОБКИ ИСКЛЮЧИТЕЛЬНО
."         HK ->    ПРИ ИСПОЛНЕНИИ НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ  28
                 СЛЕДУЮЩИЕ ЛИТЕРЫ ДО КАВЫЧКИ ИСКЛЮЧИТЕЛЬНО
.R          + N1,+N2->          НАПЕЧАТАТЬ N1 НА ТЕРМИНАЛЕ  38
                                 В ПОЛЕ ДЛИНЫ +N2 СПРАВА
.VOC        * PFA+2->         НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ ИМЯ  43
                          СЛОВАРНОЙ СТАТЬИ ДЛЯ СПИСКА СЛОВ
<             N1,N2->F         F НЕ НУЛЬ, ЕСЛИ N1 МЕНЬШЕ N2  19
<>          + W1,W2->F      F НЕ НУЛЬ, ЕСЛИ W1 НЕ РАВНО W2  19
<#            ->            НАЧАТЬ ФОРМАТНОЕ ПРЕОБРАЗОВАНИЕ  37
<MARK       K ->A  ОТМЕТИТЬ ТЕКУЩИЙ АДРЕС ДЛЯ ССЫЛКИ НАЗАД  19
<RESOLVE    K A->         РАЗРЕШИТЬ ССЫЛКУ НАЗАД В АДРЕС А  19
(           H ->  КОММЕНТАРИЙ - ПРОПУСТИТЬ СЛЕДУЮЩИЙ ТЕКСТ  28
                              ДО ЗАКРЫВАЮЩЕЙ КРУГЛОЙ СКОБКИ
(.")       *K ->             ПРОЦЕДУРА, КОМПИЛИРУЕМАЯ В ".""  28
(+LOOP)     * N->       ТЕСТ НА ЗАВЕРШЕНИЕ ЦИКЛА "DO +LOOP"   5
                                                  С ШАГОМ N
(;CODE)    *K ->      ЗАПИСАТЬ В ПОЛЕ КОДА ПОСЛЕДНЕЙ СТАТЬИ  31
                    СЛЕДУЮЩИЙ АДРЕС И ЗАКОНЧИТЬ ОПРЕДЕЛЕНИЕ
(#SCR)      * N->A,T       ПЕРЕВЕСТИ НОМЕР ЭКРАНА N В ТЕКСТ  46
(A")        * F->       ПРОЦЕДУРА, КОМПИЛИРУЕМАЯ В "ABORT""  29
(DO)       *K W1,W2-> ВХОД В ЦИКЛ СО СЧЕТЧИКОМ ОТ W2 ДО W1  24
(EXPECT)    * A,+N1->A,+N2                ВВЕСТИ С ТЕРМИНАЛА   6
                  +N1 ЛИТЕР ПО АРЕСУ А ДО ПЕРЕВОДА СТРОКИ;
                   +N2 - ФАКТИЧЕСКОЕ ЧИСЛО ВВЕДЕННЫХ ЛИТЕР
(FIND)      * -1,AN,,,A1,T->CFA,C,TF/FF      ИСКАТЬ СЛОВО T  34
                    В СПИСКАХ A1,,,AN; ПРИ УСПЕХЕ ДАТЬ CFA
                   ЕГО СТАТЬИ И С - БАЙТ ДЛИНЫ С ПРИЗНАКАМИ
(FORGET)    * A->  УДАЛИТЬ СЛОВАРНЫЕ СТАТЬИ ПОСЛЕ АДРЕСА А  45
(LOOP)      * ->         ТЕСТ НА ЗАВЕРШЕНИЕ ЦИКЛА "DO LOOP"   5
(VOC)       * PFA1+2->PFA2,N/0,N      ДАТЬ ЧИСЛО СТАТЕЙ N В  43
                      СПИСКЕ PFA1+2 И PFA СЛЕДУЮЩЕГО СПИСКА
                                       ИЛИ НУЛЬ, ЕСЛИ ЕГО НЕТ
+             W1,W2->W3                    СУММА ЧИСЕЛ W1 И W2  17
+!            W,A->     УВЕЛИЧИТЬ ЗНАЧЕНИЕ ПО АДРЕСУ А НА W  17
+BUF        * A1->A2,F  ПЕРЕЙТИ К СЛЕДУЮЩЕМУ БУФЕРУ В ПУЛЕ  25
+LOOP      HK A1,A2,3-> /КОМПИЛЯЦИЯ/            КОНЕЦ ЦИКЛА  47
              N->       /ИСПОЛНЕНИЕ/  "DO +LOOP" С ШАГОМ N
!             W,A->           ЗАСЛАТЬ ЗНАЧЕНИЕ W ПО АДРЕСУ А  11
!CSP        * ->      ЗАПОМНИТЬ АДРЕС ВЕРШИНЫ СТЕКА В "CSP"  29
]             ->        ПЕРЕКЛЮЧИТЬ ТЕКСТОВЫЙ ИНТЕРПРЕТАТОР  22
                                        В РЕЖИМ КОМПИЛЯЦИИ
*             N1,N2->N3          ПРОИЗВЕДЕНИЕ ЧИСЕЛ N1 И N2  18
*/            N1,N2,N3->N4  ЧАСТНОЕ ОТ ДЕЛЕНИЯ N1*N2 НА N3  18
*/MOD         N1,N2,N3->N4,N5      ОСТАТОК N4 И ЧАСТНОЕ N5  18
                        ОТ ДЕЛЕНИЯ ПРОИЗВЕДЕНИЯ N1*N2 НА N3
;          HK ->      ЗАКОНЧИТЬ ОПРЕДЕЛЕНИЕ ЧЕРЕЗ ДВОЕТОЧИЕ  32
;S        +H3 ->          ЗАКОНЧИТЬ ИНТЕРПРЕТАЦИЮ ЭКРАНА  41
-             W1,W2->W3                  ВЫЧЕСТЬ W2 ИЗ W1  17
-->       +H3 ->         ИНТЕРПРЕТИРОВАТЬ СЛЕДУЮЩИЙ ЭКРАН  41
-FIND       * ->A,N      ВВЕСТИ СЛОВО И ИСКАТЬ В СЛОВАРЕ;  35
                         РЕЗУЛЬТАТ ТОТ ЖЕ, ЧТО И У "FIND"
-TRAILING     A,N1->A,N2               ОТСЕЧЬ КОНЕЧНЫЕ ПРОБЕЛЫ  41
```

```
/              N1,N2->N3      ЧАСТНОЕ ОТ ДЕЛЕНИЯ N1 НА N2   18
/MOD           N1,N2->N3,N4         ОСТАТОК N3 И ЧАСТНОЕ N4   18
                                         ОТ ДЕЛЕНИЯ N1 НА N2
,              W->     СКОМПИЛИРОВАТЬ W НА ВЕРШИНУ СЛОВАРЯ    11
,"           * ->      СКОМПИЛИРОВАТЬ СТРОКУ СО СЧЕТЧИКОМ     28
>              N1,N2->F        F НЕ НУЛЬ, ЕСЛИ N1 БОЛЬШЕ N2   19
>=           * N1,N2->F     F НЕ НУЛЬ, ЕСЛИ N1 НЕ МЕНЬШЕ N2   19
>BODY          CFA->PFA       ОТ ПОЛЯ КОДА К ПОЛЮ ПАРАМЕТРОВ  30
>IN          П ->A   ПЕРЕМЕННАЯ - СМЕЩЕНИЕ ОЧЕРЕДНОЙ ЛИТЕРЫ    8
                        ВО ВХОДНОМ ТЕКСТОВОМ БУФЕРЕ ИЛИ ЭКРАНЕ
>LINK        * CFA->LFA   ПЕРЕЙТИ ОТ ПОЛЯ КОДА К ПОЛЮ СВЯЗИ   30
>MARK        K ->A ОТМЕТИТЬ ТЕКУЩИЙ АДРЕС ДЛЯ ССЫЛКИ ВПЕРЕД   19
>NAME        * CFA->NFA   ПЕРЕЙТИ ОТ ПОЛЯ КОДА К ПОЛЮ ИМЕНИ   30
>R           K W->             ПЕРЕНЕСТИ W НА СТЕК ВОЗВРАТОВ   9
>RESOLVE     K A->        РАЗРЕШИТЬ ССЫЛКУ ВПЕРЕД В АДРЕСЕ A  19
?            * A->           НАПЕЧАТАТЬ ЗНАЧЕНИЕ ПО АДРЕСУ A  38
?+           * +N->+N       ПРОВЕРИТЬ, ЧТО +N НЕОТРИЦАТЕЛЬНО  29
?ABORT       * F,T->         ЕСЛИ F НЕ НУЛЬ, ТО НАПЕЧАТАТЬ НА 29
                      ТЕРМИНАЛЕ СТРОКУ T И УЙТИ НА "ABORT"
?BRANCH      K F->    ЕСЛИ F "ЛОЖЬ", ТО КАК "BRANCH", ИНАЧЕ    5
                      ПРОДОЛЖИТЬ ИНТЕРПРЕТАЦИЮ ОТ АДРЕСА,
                             СЛЕДУЮЩЕГО ЗА АДРЕСОМ ПЕРЕХОДА
?COMP        * -> ПРОВЕРИТЬ, ЧТО ТЕКУЩИЙ РЕЖИМ - КОМПИЛЯЦИЯ   29
?CSP         * ->     ВЫДАТЬ ОШИБКУ "СБИЛСЯ УКАЗАТЕЛЬ СТЕКА"  29
                          ЕСЛИ ОН НЕ РАВЕН ЗНАЧЕНИЮ В "CSP"
?DUP           W->W,W   ПРОДУБЛИРОВАТЬ W, ЕСЛИ ЭТО НЕ НУЛЬ     9
?GAP         * N->  ВЫДАТЬ ОШИБКУ "ИСЧЕРПАНИЕ ПАМЯТИ", ЕСЛИ   29
                     ЗАЗОР МЕЖДУ ВЕРШИНАМИ СТЕКА И СЛОВАРЯ
                                            МЕНЕЕ N БАЙТОВ
?LOADING     * ->     ВЫДАТЬ ОШИБКУ "НЕТ ОБРАБОТКИ ЭКРАНА",   29
                      ЕСЛИ ВХОДНОЙ ТЕКСТ ИДЕТ НЕ С ЭКРАНА
?PAIRS       * W1,W2->    ВЫДАТЬ ОШИБКУ, "НЕПАРНЫЕ СКОБКИ",   29
                                       ЕСЛИ W1 НЕ РАВНО W2
?STACK       * ->    ВЫДАТЬ ОШИБКУ "ИСЧЕРПАНИЕ СТЕКА", ЕСЛИ   29
                 ОН БОЛЕЕ, ЧЕМ ПУСТ, И "ИСЧЕРПАНИЕ ПАМЯТИ"
                                ПРИ ЗАЗОРЕ, МЕНЬШЕМ 10 БАЙТОВ
:              ->  НАЧАТЬ ОПРЕДЕЛЕНИЕ СЛОВА ЧЕРЕЗ ДВОЕТОЧИЕ   32
#              D1->D2        ДЕЛЕНИЕМ D1 НА ЗНАЧЕНИЕ "BASE"   37
                       ВЫДЕЛИТЬ 1 ЦИФРУ С КОНЦА И ДОБАВИТЬ ЕЕ
                          В БУФЕР "PAD", ОСТАВИВ ЧАСТНОЕ D2
#>             D->A,+N  ЗАКОНЧИТЬ ФОРМАТНОЕ ПРЕОБРАЗОВАНИЕ;   37
                      ДАТЬ АДРЕС A НАЧАЛА ЛИТЕР И ИХ ЧИСЛО +N
#S             D1->0,0   ВЫДЕЛЯТЬ ЦИФРЫ D1 ПО СЛОВУ "#" ДО    37
                                            ПОЛУЧЕНИЯ НУЛЯ
#TIB         П ->A  ПЕРЕМЕННАЯ - ЧИСЛО ЛИТЕР В БУФЕРЕ "TIB"    8
@              A->W              ДАТЬ ЗНАЧЕНИЕ ПО АДРЕСУ A    11
'              ->CFA          ДАТЬ CFA ДЛЯ СЛЕДУЮЩЕГО СЛОВА   41
=              W1,W2->F        F НЕ НУЛЬ, ЕСЛИ W1 РАВНО W2    19
"           *HK -> /КОМПИЛЯЦИЯ/    СКОМПИЛИРОВАТЬ СЛЕДУЮЩИЕ   28
               ->T /ИСПОЛНЕНИЕ/            ЛИТЕРЫ ДО КАВЫЧКИ
                      ИСКЛЮЧИТЕЛЬНО КАК СТРОКУ СО СЧЕТЧИКОМ
".           * T->        НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ СТРОКУ T    28
ABORT                       СБРОСИТЬ СТЕК И УЙТИ ПО "QUIT"    28
ABORT"      KH -> /КОМПИЛЯЦИЯ/  ЕСЛИ F "ИСТИНА" (НЕ НУЛЬ)     29
               F-> /ИСПОЛНЕНИЕ/  ТО НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ
               СЛЕДУЮЩИЙ ТЕКСТ ДО КАВЫЧКИ И УЙТИ НА "ABORT"
ABORT8       * ->     ВЫДАТЬ ОШИБКУ "НЕПРАВИЛЬНОЕ ЗНАЧЕНИЕ"   29
ABS            N1->N2                    АБСОЛЮТНАЯ ВЕЛИЧИНА  17
AGAIN       +HK A,1-> /КОМПИЛЯЦИЯ/ КОНЕЦ ЦИКЛА "BEGIN AGAIN"  47
               ->     /ИСПОЛНЕНИЕ/
ALIGN        * +N->         ВЫРОВНЯТЬ ВЕРШИНУ СЛОВАРЯ НА +N   10
ALIGNH       * ->     ВЫРОВНЯТЬ ВЕРШИНУ СЛОВАРЯ НА ПОЛУСЛОВО  10
ALLOT          W->      СМЕСТИТЬ ВЕРШИНУ СЛОВАРЯ НА W БАЙТОВ  10
ALPHA        * N->C      ПРЕОБРАЗОВАТЬ N В ЛИТЕРУ C КАК ЦИФР  37
```

```
AND            W1,W2->W3           ПОРАЗРЯДНОЕ ЛОГИЧЕСКОЕ "И"   13
B/BUF        + ->1024         ЧИСЛО БАЙТОВ В БЛОЧНОМ БУФЕРЕ    7
BADWORD      * A->            СООБЩИТЬ О НЕОПОЗНАННОМ СЛОВЕ   29
BASE         П ->A   ПЕРЕМЕННАЯ - ТЕКУЩЕЕ ОСНОВАНИЕ СИСТЕМЫ    8
                          СЧИСЛЕНИЯ ПРИ ВВОДЕ-ВЫВОДЕ ЧИСЕЛ
BEGIN       НК ->A,1 /КОМПИЛЯЦИЯ/       НАЧАЛО ЦИКЛА "BEGIN"   47
               ->    /ИСПОЛНЕНИЕ/
BL           + ->64          КОНСТАНТА - КОД ПРОБЕЛА В ДКОИ    7
BLANK        + A,U-> ЗАСЛАТЬ ПРОБЕЛЫ В U БАЙТОВ ПО АДРЕСУ A   22
BLK          П ->A ПЕРЕМЕННАЯ - НОМЕР ВХОДНОГО БЛОКА-ЭКРАНА    8
BLOCK          +N->A        ДАТЬ АДРЕС А БУФЕРА С БЛОКОМ +N   25
BODY>        * PFA->CFA     ОТ ПОЛЯ ПАРАМЕТРОВ К ПОЛЮ КОДА    30
BRANCH       K ->       ПРОДОЛЖИТЬ ИНТЕРПРЕТАЦИЮ ОТ ЗНАЧЕНИЯ   5
                         СЛЕДУЮЩЕГО СКОМПИЛИРОВАННОГО АДРЕСА
BRANCH#      M         ПРОДОЛЖЕНИЕ ИНТЕРПРЕТАЦИИ ОТ АДРЕСА В   5
                                          СЛЕДУЮЩЕМ ПОЛУСЛОВЕ
BUFFER         +N->A                 ПРИПИСАТЬ БЛОКУ +N БУФЕР  25
C!             C,A->               ЗАСЛАТЬ БАЙТ С ПО АДРЕСУ А  11
C,           + C-> СКОМПИЛИРОВАТЬ БАЙТ С НА ВЕРШИНУ СЛОВАРЯ    11
C@             A->C                     ДАТЬ БАЙТ ПО АДРЕСУ А  11
C"          *Н ->  /КОМПИЛЯЦИЯ/   СКОМПИЛИРОВАТЬ КОД ПЕРВОЙ    28
               ->C /ИСПОЛНЕНИЕ/      ЛИТЕРЫ СЛЕДУЮЩЕГО СЛОВА
                                                 КАК ЛИТЕРАЛ
CMOVE          A1,A2,U->      ПЕРЕСЛАТЬ U БАЙТОВ ОТ A1 В A2    21
CMOVE>         A1,A2,U->    ПЕРЕСЛАТЬ U БАЙТОВ ОТ АДРЕСА A1    21
                       ПО АДРЕСУ A2 НАЧИНАЯ С БОЛЬШИХ АДРЕСОВ
COMPILE      K ->             КОМПИЛИРОВАТЬ СЛЕДУЮЩИЙ АДРЕС    22
CONSTANT       W->               ОПРЕДЕЛИТЬ СЛЕДУЮЩЕЕ СЛОВО    32
                              КАК КОНСТАНТУ СО ЗНАЧЕНИЕМ W
CONTEXT      П ->A          ПЕРЕМЕННАЯ - СПИСОК, С КОТОРОГО     7
                              НАЧИНАЕТСЯ ПОИСК ВВОДИМЫХ СЛОВ
CONVERT        WD1,A1,->WD2,A2   ПРЕОБРАЗОВАТЬ WD1 И ЛИТЕРЫ    39
                  ОТ A1+1 В WD2 И A2 - АДРЕС 1-ОЙ НЕ ЦИФРЫ
COUNT          T->A,N      ДАТЬ АДРЕС ПЕРВОЙ ЛИТЕРЫ И ЧИСЛО    28
                              ЛИТЕР N СТРОКИ СО СЧЕТЧИКОМ Т
CR             ->         ВЫВЕСТИ НА ТЕРМИНАЛ ПЕРЕВОД СТРОКИ    6
CREATE         ->         СОЗДАТЬ НАЧАЛО СТАТЬИ (ДО PFA) ДЛЯ   36
                                             СЛЕДУЮЩЕГО СЛОВА;
                           ЕГО ИСПОЛНЕНИЕ КЛАДЕТ PFA НА СТЕК
CREATE#      A         НАЧАЛО ИСПОЛНИТЕЛЬНОЙ ЧАСТИ "VARIABLE"   3
CSP         *П ->A      ПЕРЕМЕННАЯ ДЛЯ КОНТРОЛЬНОГО ХРАНЕНИЯ    8
                                    ЗНАЧЕНИЯ УКАЗАТЕЛЯ СТЕКА
CURRENT      П ->A  ПЕРЕМЕННАЯ - СПИСОК ДЛЯ ДОБАВЛЕНИЙ СЛОВ     7
D.             D->  НАПЕЧАТАТЬ D НА ТЕРМИНАЛЕ И ДАТЬ ПРОБЕЛ    38
D.R            D,+N->   НАПЕЧАТАТЬ D В ПОЛЕ ДЛИНЫ +N СПРАВА   38
D<             D1,D2->F        F "ИСТИНА", ЕСЛИ D1 МЕНЬШЕ D2   15
D+             WD1,WD2->WD3   СУММА ДВОЙНЫХ ЧИСЕЛ WD1 И WD2    14
D-             WD1,WD2->WD3  РАЗНОСТЬ ДВОЙНЫХ ЧИСЕЛ WD1-WD2    14
D/           * D1,D2->D3      ЧАСТНОЕ D3 ОТ ДЕЛЕНИЯ D1 НА D2   15
D/MOD        * D1,D2->D3,D4        ОСТАТОК D3 И ЧАСТНОЕ D4 ОТ  15
                             ДЕЛЕНИЯ ДВОЙНЫХ ЧИСЕЛ D1 НА D2
D=             WD1,WD2->F  F "ИСТИНА", ЕСЛИ WD1 И WD2 РАВНЫ    15
DABS           D1->D2    АБСОЛЮТНАЯ ВЕЛИЧИНА ДВОЙНОГО ЧИСЛА    14
DECIMAL        ->              ПЕРЕЙТИ В ДЕСЯТИЧНУЮ СИСТЕМУ    22
DEFINITIONS ->  УСТАНОВИТЬ СПИСОК "CURRENT" НА "CONTEXT"       31
DEPTH          ->+N       КОЛИЧЕСТВО ЗНАЧЕНИЙ НА СТЕКЕ ДО +N   20
DIGIT        * C,N1->N2,TF/FF      N2 - ЗНАЧЕНИЕ ЛИТЕРЫ С КАК  39
             ЦИФРЫ В СИСТЕМЕ СЧИСЛЕНИЯ ПО ОСНОВАНИЮ N1
DMAX           WD1,WD2->WD3           БОЛЬШЕЕ ИЗ ДВУХ ЧИСЕЛ    16
DMIN           WD1,WD2->WD3           МЕНЬШЕЕ ИЗ ДВУХ ЧИСЕЛ    16
DMOD         * D1,D2->D3      ОСТАТОК D3 ОТ ДЕЛЕНИЯ D1 НА D2   15
DNEGATE        D1->D2         РЕЗУЛЬТАТ ВЫЧИТАНИЯ D1 ИЗ НУЛЯ   14
DO          НК ->A1,A2,3 /КОМПИЛЯЦИЯ/  НАЧАЛО ЦИКЛА "DO" СО    47
               N1,N2->   /ИСПОЛНЕНИЕ/ СЧЕТЧИКОМ ОТ N2 ДО N1
```

```
DOES>      НК ->  НАЧАЛО "ИСПОЛНЕНИЯ" В ОПРЕДЕЛЯЮЩЕМ СЛОВЕ  36
DOES#      М      ПОДПРОГРАММА - НАЧАЛО РАСШИРЕНИЯ "DOES>"   3
DP!        * А->     УСТАНОВИТЬ ВЕРШИНУ СЛОВАРЯ НА АДРЕС А  10
DPL        +П ->А      ПЕРЕМЕННАЯ - ПОЗИЦИЯ ПОСЛЕДНЕЙ ТОЧКИ  8
                       В ПОСЛЕДНЕМ ВВЕДЕННОМ ЧИСЛЕ ОТ КОНЦА
DROP          W->          УБРАТЬ СО СТЕКА ВЕРХНЕЕ ЗНАЧЕНИЕ  9
DU<           UD1,UD2->F   F "ИСТИНА", ЕСЛИ UD1 МЕНЬШЕ UD2  14
DUMP       + A,U->      РАСПЕЧАТАТЬ НА ТЕРМИНАЛЕ U БАЙТОВ  42
                                                ОТ АДРЕСА А
DUP           W->W,W       ПРОДУБЛИРОВАТЬ ВЕРХНЕЕ ЗНАЧЕНИЕ  9
D0<           D->F           F "ИСТИНА", ЕСЛИ D МЕНЬШЕ НУЛЯ  15
D0=           WD->F                F "ИСТИНА", ЕСЛИ WD НУЛЬ  15
D2/           D1->D2                      РАЗДЕЛИТЬ НА ДВА  15
ELSE       НК A1,2->A2,2 /КОМПИЛЯЦИЯ/    НАЧАЛО 2-ОЙ ВЕТВИ  47
              ->         /ИСПОЛНЕНИЕ/        ВЕТВЛЕНИЯ "IF"
EMIT          C->     ВЫВЕСТИ НА ТЕРМИНАЛ ЛИТЕРУ С КОДОМ С  6
EMPTY-BUFFERS + ->                 ОЧИСТИТЬ БУФЕРНЫЙ ПУЛ  25
ENCLOSE    * A,C->A,N1,N2,N3                     ВВОД СЛОВА  27
ERASE      + A,U->    ЗАСЛАТЬ НУЛИ В U БАЙТОВ ПО АДРЕСУ А  22
ERCOND#    М         СИГНАЛИЗАЦИЯ О НЕПРАВИЛЬНОМ ЗНАЧЕНИИ  4
EXECUTE       CFA->      ИСПОЛНИТЬ СЛОВО ПО CFA ЕГО СТАТЬИ  11
EXIT       K -> ЗАКОНЧИТЬ ИСПОЛНЕНИЕ ТЕКУЩЕГО ОПРЕДЕЛЕНИЯ  4
EXIT#      М        ТОЧКА "EXIT" АДРЕСНОГО ИНТЕРПРЕТАТОРА  4
                                         ЗНАЧЕНИИ В СТЕКЕ
EXPECT        A,+N->        ВВЕСТИ С ТЕРМИНАЛА +N ЛИТЕР ПО  40
                    АДРЕСУ А; В ПЕРЕМЕННУЮ "SPAN" ЗАСЛАТЬ
                       ФАКТИЧЕСКОЕ ЧИСЛО ВВЕДЕННЫХ ЛИТЕР;
                             ЛИТЕРЫ НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ
FENCE      *П ->А  ПЕРЕМЕННАЯ - ГРАНИЦА ЗАЩИТЫ ОТ "FORGET"  7
FILL          A,U,C->     ЗАСЛАТЬ С В U БАЙТОВ ПО АДРЕСУ А  22
FIND          T->A,N    ИСКАТЬ СЛОВО Т В ТЕКУЩЕМ КОНТЕКСТЕ  35
                    ЕСЛИ N=0, ТО А=Т И СЛОВО НЕ НАЙДЕНО;
                   ИНАЧЕ A=CFA НАЙДЕННОЙ СТАТЬИ, N=1 ДЛЯ
                   СЛОВ "IMMEDIATE" И N=-1 ДЛЯ ОСТАЛЬНЫХ
FIRST      * ->А  КОНСТАНТА - АДРЕС НАЧАЛА БУФЕРНОГО ПУЛА  2
FIRST#     М                  ЗНАЧЕНИЕ КОНСТАНТЫ "FIRST"  2
FL#        А    ПОЛЕ СВЯЗИ ДЛЯ СПИСКОВ В ПОЛЕ ПАРАМЕТРОВ  33
                        СЛОВАРНОЙ СТАТЬИ СЛОВА "FORTH"
FLUSH         ->     ЗАПИСАТЬ БЛОКИ НА ДИСК И ОЧИСИТИТЬ ПУЛ  26
FORGET        -> УДАЛИТЬ СЛОВАРНУЮ СТАТЬЮ СЛЕДУЮЩЕГО СЛОВА  45
                     И ВСЕХ СЛОВ, ОПРЕДЕЛЕННЫХ ПОСЛЕ НЕГО
FORTH         ->  УСТАНОВИТЬ "CONTEXT" НА НАЧАЛЬНЫЙ СПИСОК  33
FORTH-83      ->          СТАНДАРТНЫЙ КОНТЕКСТ ФОРТ-СИСТЕМЫ  33
FORTH#     А               PFA+2 ДЛЯ СЛОВАРНОЙ СТАТЬИ "FORTH"  33
GOTO       М                 ПОДПРОГРАММА ПЕРЕХОДА ПО ССЫЛКЕ  4
H.         + U->                   НАПЕЧАТАТЬ U НА ТЕРМИНАЛЕ  38
                           В 16-НОЙ СИСТЕМЕ И ДАТЬ ПРОБЕЛ
HERE          ->А        ДАТЬ АДРЕС ТЕКУЩЕЙ ВЕРШИНЫ СЛОВАРЯ  10
HEX        + ->         ПЕРЕЙТИ В ШЕСТНАДЦАТИРИЧНУЮ СИСТЕМУ  22
HLD        *П ->А   ПЕРЕМЕННАЯ - ПОЗИЦИЯ ПОСЛЕДНЕЙ ЛИТЕРЫ,  8
                     ПЕРЕНЕСЕННОЙ В БУФЕР "PAD" ПО "HOLD"
HOLD          C-> ПЕРЕНЕСТИ ЛИТЕРУ С НА ВЕРШИНУ БУФЕРА PAD  37
I          K ->W    ТЕКУЩЕЕ ЗНАЧЕНИЕ W СЧЕТЧИКА ЦИКЛА "DO"  24
I'         +K ->W  КОНЕЧНОЕ ЗНАЧЕНИЕ W СЧЕТЧИКА ЦИКЛА "DO"  24
ID.        *  NFA->     НАПЕЧАТАТЬ ИМЯ СЛОВА И ДАТЬ ПРОБЕЛ  31
IF         НК ->A,2 /КОМПИЛЯЦИЯ/     НАЧАЛО ВЕТВЛЕНИЯ "IF"  47
              F->   /ИСПОЛНЕНИЕ/
IMMEDIATE     ->        ДАТЬ ПРИЗНАК "IMMEDIATE" ПОСЛЕДНЕЙ  31
                                СОЗДАННОЙ СЛОВАРНОЙ СТАТЬЕ
INDEX      + N1,N2-> РАСПЕЧАТАТЬ НАЧАЛЬНУЮ СТРОКУ ЭКРАНОВ  46
                                      С НОМЕРАМИ ОТ N1 ДО N2
INTERPRET + ->              ИНТЕРПРЕТИРОВАТЬ ВХОДНОЙ ПОТОК  40
IPUSH      М   ПОДПРОГРАММА - ПОМЕСТИТЬ НА СТЕК УКАЗАТЕЛЬ  4
                  ИНТЕРПРЕТАЦИИ И ОБОЙТИ СЛЕДУЮЩУЮ СТРОКУ
```

```
J          K ->W      ТЕКУЩЕЕ ЗНАЧЕНИЕ И СЧЕТЧИКА ВТОРОГО  24
                                  ОБ"ЕМЛЮЩЕГО ЦИКЛА "DO"
KEY          ->C              ВВЕСТИ ЛИТЕРУ С ТЕРМИНАЛА   6
L>NAME     * LFA->NFA  ПЕРЕЙТИ ОТ ПОЛЯ СВЯЗИ К ПОЛЮ ИМЕНИ  30
LATEST     * ->NFA    ДАТЬ NFA ПОСЛЕДНЕЙ СОЗДАННОЙ СТАТЬИ  31
LEAVE      K ->            ЗАКОНЧИТЬ ИСПОЛНЕНИЕ ЦИКЛА "DO"  24
LENGMASK   M              ПОЛНОЕ СЛОВО - МАСКА ДЛЯ УДАЛЕНИЯ   2
                          БИТА "IMMEDIATE" ИЗ БАЙТА ДЛИНЫ
LENG1MSK   M              ПОЛНОЕ СЛОВО - МАСКА ДЛЯ УДАЛЕНИЯ   2
             БИТОВ "IMMEDIATE" И "SMUDGE" ИЗ БАЙТА ДЛИНЫ
LENG2MSK   M              ПОЛНОЕ СЛОВО - МАСКА ДЛЯ ВЫСЕЧЕНИЯ  2
                 ЧИСТОЙ ДЛИНЫ ИЗ БАЙТА ДЛИНЫ С ПРИЗНАКАМИ
LHRW12     M  ПОДПРОГРАММА ЗАГРУЗКИ ДВУХ ВЕРХНИХ ЗНАЧЕНИЙ   4
                  НА СТЕКЕ В РЕГИСТРЫ RW2 (ВЕРХНЕЕ) И RW1
LIMIT      * ->A   КОНСТАНТА - АДРЕС КОНЦА БУФЕРНОГО ПУЛА   2
LIMIT#     M                  ЗНАЧЕНИЕ КОНСТАНТЫ "LIMIT"   2
LINK>      * LFA->CFA  ПЕРЕЙТИ ОТ ПОЛЯ СВЯЗИ К ПОЛЮ КОДА  30
LIST       + N->         РАСПЕЧАТАТЬ НА ТЕРМИНАЛЕ ЭКРАН N  46
LIT       *K ->W           ПОМЕСТИТЬ НА СТЕК СЛЕДУЮЩИЙ КОД  23
LIT"      *K ->T     ДАТЬ АДРЕС СКОМПИЛИРОВАННОЙ СТРОКИ И  28
                     ПРОДОЛЖИТЬ ИНТЕРПРЕТАЦИЮ, ОБОЙДЯ ЕЕ
LITERAL    H W-> /КОМПИЛЯЦИЯ/          СКОМПИЛИРОВАТЬ W КАК  23
             ->W /ИСПОЛНЕНИЕ/                        ЛИТЕРАЛ
LOAD        +N->     ИНТЕРПРЕТИРОВАТЬ ЭКРАН С НОМЕРОМ +N  41
LOOP      HK A1,A2,3-> /КОМПИЛЯЦИЯ/ КОНЕЦ ЦИКЛА "DO LOOP"  47
             ->          /ИСПОЛНЕНИЕ/
LRW1       M   ПОДПРОГРАММА ЗАГРУЗКИ ДВОЙНОГО ЗНАЧЕНИЯ НА   4
                               ВЕРШИНЕ СТЕКА В РЕГИСТР RW1
LRW12      M   ПОДПРОГРАММА ЗАГРУЗКИ ДВУХ ВЕРХНИХ ДВОЙНЫХ   4
        ЗНАЧЕНИЙ НА СТЕКЕ В РЕГИСТРЫ RW2 (ВЕРХНЕЕ) И RW1
M*         * N1,N2->D  ПРОИЗВЕДЕНИЕ ДВОЙНОЙ ДЛИНЫ N1 И N2  18
M/         * D,N1->N2,N3        ОСТАТОК N2 И ЧАСТНОЕ N3 ОТ  18
                        ДЕЛЕНИЯ ДВОЙНОГО D НА ОДИНАРНОЕ N1
M/MOD      * UD1,U2->U3,UD4            ОСТАТОК U3 И ДВОЙНОЕ  16
                       ЧАСТНОЕ UD4 ОТ ДЕЛЕНИЯ UD1 НА U2
MAX          N1,N2->N3             БОЛЬШЕЕ ИЗ ЧИСЕЛ N1 И N2  22
MIN          N1,N2->N3             МЕНЬШЕЕ ИЗ ЧИСЕЛ N1 И N2  22
MOD          N1,N2->N3          ОСТАТОК ОТ ДЕЛЕНИЯ N1 НА N2  18
MSG        * ->A     КОНСТАНТА - АДРЕС НАЧАЛА БУФЕРА MSG   2
MSG#       M                    ЗНАЧЕНИЕ КОНСТАНТЫ "MSG"   2
N>LINK     * NFA->LFA  ПЕРЕЙТИ ОТ ПОЛЯ ИМЕНИ К ПОЛЮ СВЯЗИ  30
NAME>      * NFA->CFA   ПЕРЕЙТИ ОТ ПОЛЯ ИМЕНИ К ПОЛЮ КОДА  30
NEGATE       W1->W2         РЕЗУЛЬТАТ ВЫЧИТАНИЯ W1 ИЗ НУЛЯ  17
NEXT       M                 ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР   1
NEXT1      M        ПРОДОЛЖЕНИЕ АДРЕСНОЙ ИНТЕРПРЕТАЦИИ ОТ   1
                               ФОРТ-АДРЕСА В РЕГИСТРЕ 14
NOT          W1->W2              ПОРАЗРЯДНОЕ ИНВЕРТИРОВАНИЕ  13
NUMBER     + T->WD      ПРЕОБРАЗОВАТЬ СТРОКУ T В ЧИСЛО WD  39
OFFSET    +П ->A      ПЕРЕМЕННАЯ - ДОБАВКА К НОМЕРУ БЛОКА   8
OR           W1,W2->W3        ПОРАЗРЯДНОЕ ЛОГИЧЕСКОЕ "ИЛИ"  13
OVER         W1,W2->W1,W2,W1 ПРОДУБЛИРОВАТЬ ВТОРОЕ СВЕРХУ   9
PAD          ->A    ДАТЬ АДРЕС ТЕКУЩЕЙ ВЕРШИНЫ БУФЕРА PAD  37
PICK         WN,...,WO,+N->WN,...,WO,WN     ПРОДУБЛИРОВАТЬ  12
                                     N-Е СВЕРХУ ЗНАЧЕНИЕ
POP        M     ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР СО СНЯТИЕМ   3
                     ВЕРХНЕГО ЗНАЧЕНИЯ С ВЕРШИНЫ СТЕКА
POPPUT1    M     ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР СО СНЯТИЕМ   3
            ВЕРХНЕГО И ЗАМЕНОЙ ПРЕДЫДУЩЕГО НА ЗНАЧЕНИЕ
                                        ИЗ РЕГИСТРА RW1
PREV      *C ->A     ПЕРЕМЕННАЯ - ТЕКУЩИЙ БЛОЧНЫЙ БУФЕР   7
PUSHRW1    M   ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР С ПОМЕЩЕНИЕМ   3
              ЗНАЧЕНИЯ ИЗ РЕГИСТРА RW1 НА ВЕРШИНУ СТЕКА
PUSH2RW1   M      ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР С ЗАМЕНОЙ   3
           ВЕРХНЕГО НА ДВОЙНОЕ ЗНАЧЕНИЕ ИЗ РЕГИСТРА RW1
```

```
PUTRW1     M       ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР С ЗАМЕНОЙ    3
                      ВЕРХНЕГО ЗНАЧЕНИЯ НА ЗНАЧЕНИЕ ИЗ RW1
QUERY      + ->  ВВЕСТИ С ТЕРМИНАЛА ЛИТЕРЫ В БУФЕР "TIB";    40
                    ЧИСЛО ВВЕДЕННЫХ ЛИТЕР ЗАСЛАТЬ В "#TIB"
QUIT             СБРОСИТЬ СТЕК ВОЗВРАТОВ, ПЕРЕЙТИ В РЕЖИМ    28
                   ИСПОЛНЕНИЯ И ПРОДОЛЖИТЬ ИНТЕРПРЕТАЦИЮ
R.         * ->   РАСПЕЧАТАТЬ НА ТЕРМИНАЛЕ СТЕК ВОЗВРАТОВ    42
R>         K ->W    ПЕРЕНЕСТИ ЗНАЧЕНИЕ СО СТЕКА ВОЗВРАТОВ     9
R@         K ->W     СКОПИРОВАТЬ ВЕРШИНУ СТЕКА ВОЗВРАТОВ      9
RBLK       * A,+N->        ПРОЧЕСТЬ ЭКРАН +N ПО АДРЕСУ A      6
RDROP     *K ->          СНЯТЬ ЗНАЧЕНИЕ СО СТЕКА ВОЗВРАТОВ    9
RECURSE +HK -> СКОМПИЛИРОВАТЬ ОБРАЩЕНИЕ К КОМПИЛИРУЕМОМУ    31
                            В ДАННЫЙ МОМЕНТ ОПРЕДЕЛЕНИЮ
REMEMBER  + ->     ОПРЕДЕЛИТЬ СЛОВО, ИСПОЛНЕНИЕ КОТОРОГО     45
                     УНИЧТОЖАЕТ ВСЕ ПОСЛЕДУЮЩИЕ ОПРЕДЕЛЕНИЯ
REPEAT    HK A1,1,A2,2-> /КОМПИЛЯЦИЯ/          КОНЕЦ ЦИКЛА   47
             ->    /ИСПОЛНЕНИЕ/      "BEGIN WHILE REPEAT"
ROLL         WN,WN-1,...,W0,+N->WN-1,...W0,WN  ЦИКЛИЧЕСКИ   12
                             ПЕРЕСТАВИТЬ N ВЕРХНИХ ЗНАЧЕНИЙ
ROT          W1,W2,W3->W2,W3,W1             ПЕРЕСТАВИТЬ ТРИ   9
                     ВЕРХНИХ ЗНАЧЕНИЯ ПО ЧАСОВОЙ СТРЕЛКЕ
RP!        * A->       УСТАНОВИТЬ УКАЗАТЕЛЬ ВЕРШИНЫ СТЕКА    20
                                            ВОЗВРАТОВ НА A
RP@        * ->A    АДРЕС ТЕКУЩЕЙ ВЕРШИНЫ СТЕКА ВОЗВРАТОВ    20
R0        *C ->A   ПЕРЕМЕННАЯ - АДРЕС ДНА СТЕКА ВОЗВРАТОВ     7
S.         * ->       РАСПЕЧАТАТЬ НА ТЕРМИНАЛЕ СТЕК ДАННЫХ   42
S>D        * N->D    РАСШИРИТЬ N ДО ЧИСЛА ДВОЙНОЙ ДЛИНЫ D    14
SAVE-BUFFERS ->  ЗАПИСАТЬ НА ДИСК ВСЕ ИСПРАВЛЕННЫЕ БЛОКИ    26
SCR       +П ->A      ПЕРЕМЕННАЯ - НОМЕР ЭКРАНА В "LIST"      8
SIGN         N->          ДОБАВИТЬ В ФОРМАТНУЮ СТРОКУ ЗНАК   37
                           МИНУС, ЕСЛИ ЧИСЛО N ОТРИЦАТЕЛЬНО
SMUDGE     * ->       УСТАНОВИТЬ В ЕДИНИЦУ ФЛАГ "SMUDGE"    31
                               В ПОСЛЕДНЕЙ СОЗДАННОЙ СТАТЬЕ
SNAPSTK    * A1,A2,A3->       РАСПЕЧАТКА СТЕКА ОТ A1 ДО A2   42
                      С ТЕКСТОМ A3; ВОЗВРАТ "ЧЕРЕЗ ОДИН"
SP!        * A->  УСТАНОВИТЬ УКАЗАТЕЛЬ ВЕРШИНЫ СТЕКА НА A    20
SP@        + ->A       АДРЕС ТЕКУЩЕЙ ВЕРШИНЫ СТЕКА ДАННЫХ    20
SPACE        ->            НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ ПРОБЕЛ    23
SPACES       +N->     НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ +N ПРОБЕЛОВ    23
SPAN       П ->A       ПЕРЕМЕННАЯ ДЛЯ РЕЗУЛЬТАТА "EXPECT"     8
STATE      П ->A       ПЕРЕМЕННАЯ С СОСТОЯНИЕМ ТЕКСТОВОГО     8
                  ИНТЕРПРЕТАТОРА: "ИСТИНА" - КОМПИЛЯЦИЯ
SWAP         W1,W2->W2,W1       ОБМЕНЯТЬ МЕСТАМИ 2 ВЕРХНИХ    9
S0        +П ->A      ПЕРЕМЕННАЯ - АДРЕС ДНА СТЕКА ДАННЫХ     7
TEMP       M           РАБОЧАЯ ОБЛАСТЬ ИЗ ДВУХ ДВОЙНЫХ СЛОВ   2
THEN      HK A,2-> /КОМПИЛЯЦИЯ/        КОНЕЦ ВЕТВЛЕНИЯ "IF"   47
             ->    /ИСПОЛНЕНИЕ/
THRU       + +N1,+N2-> ИНТЕРПРЕТИРОВАТЬ ЭКРАНЫ С НОМЕРАМИ    41
                             ОТ +N1 ДО +N2 ВКЛЮЧИТЕЛЬНО
TIB          ->A         АДРЕС ВХОДНОГО ТЕКСТОВОГО БУФЕРА     2
                                     ДЛЯ ВВОДА С ТЕРМИНАЛА
TIB#       M                   ФОРТ-АДРЕС НАЧАЛА БУФЕРА TIB   2
TYPE         A,+N->      НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ +N ЛИТЕР     6
                                               ОТ АДРЕСА A
U.           U->             НАПЕЧАТАТЬ U НА ТЕРМИНАЛЕ КАК   38
                                            ЧИСЛО БЕЗ ЗНАКА
U.R        + U,+N->       НАПЕЧАТАТЬ НА ТЕРМИНАЛЕ ЧИСЛО U    38
                                   В ПОЛЕ ДЛИНЫ +N СПРАВА
U<           U1,U2->F           F ИСТИНА, ЕСЛИ U1 МЕНЬШЕ U2   16
UM*          U1,U2->UD     ПРОИЗВЕДЕНИЕ UD ЧИСЕЛ U1 И U2     16
UM/MOD       UD,U1->U2,U3        ОСТАТОК U2 И ЧАСТНОЕ U3 ОТ   16
                                         ДЕЛЕНИЯ UD НА U1
UNSMUDGE   * ->          УСТАНОВИТЬ В НУЛЬ ФЛАГ "SMUDGE"    31
                               В ПОСЛЕДНЕЙ СОЗДАННОЙ СТАТЬЕ
```

UNTIL	НК	A,1-> /КОМПИЛЯЦИЯ/ F-> /ИСПОЛНЕНИЕ/	КОНЕЦ ЦИКЛА "BEGIN UNTIL"	47
UPDATE		->	ОТМЕТИТЬ ТЕКУЩИЙ БЛОК КАК ИЗМЕНЕННЫЙ	25
USE	*C	->A	ПЕРЕМЕННАЯ - СЛЕДУЮЩИЙ БЛОЧНЫЙ БУФЕР	7
VARIABLE		->	ОПРЕДЕЛИТЬ СЛЕДУЮЩЕЕ СЛОВО КАК ПЕРЕМЕННУЮ С НАЧАЛЬНЫМ ЗНАЧЕНИЕМ НУЛЬ	32
VOC-LINK	*П	->A	ПЕРЕМЕННАЯ - АДРЕС ПОЛЯ СВЯЗИ ПОСЛЕДНЕГО СОЗДАННОГО ПО "VOCABULARY" СПИСКА СЛОВ	33
VOCABULARY		->	ОПРЕДЕЛИТЬ СЛЕДУЮЩЕЕ СЛОВО КАК СПИСОК НАД ТЕКУЩИМ ЗНАЧЕНИЕМ "CURRENT"	33
VOCABULARY#		A	НАЧАЛО ИСПОЛНИТЕЛЬНОЙ ЧАСТИ "VOCABULARY"	33
VOCS	*	->	РАСПЕЧАТАТЬ НА ТЕРМИНАЛЕ ТЕКУЩИЙ ПОРЯДОК ПОИСКА СЛОВ В СЛОВАРЕ	43
WBLK	*	A,+N->	ЗАПИСАТЬ ЭКРАН +N ИЗ АДРЕСА A	6
WHILE	НК	1->A,2 /КОМПИЛЯЦИЯ/ F-> /ИСПОЛНЕНИЕ/	ВЕТВЛЕНИЕ "WHILE" В ЦИКЛЕ "BEGIN WHILE REPEAT"	47
WIDTH	*	->N	КОНСТАНТА - МАКСИМАЛЬНАЯ ДЛИНА ИМЕНИ	7
WORD		C->T	ВВЕСТИ СЛОВО ДО СТОП-ЛИТЕРЫ C; ДАТЬ ЕГО АДРЕС КАК СТРОКИ СО СЧЕТЧИКОМ	27
WORDS	*	->	РАСПЕЧАТАТЬ НА ТЕРМИНАЛЕ ИМЕНА СЛОВ ИЗ СПИСКА "CONTEXT"	44
XOR		W1,W2->W3	ПОРАЗРЯДНОЕ "ИСКЛЮЧАЮЩЕЕ ИЛИ"	13
0	*	->0	КОНСТАНТА НУЛЬ (ЗНАЧЕНИЕ "ЛОЖЬ")	7
0<		N->F	F "ИСТИНА", ЕСЛИ N ОТРИЦАТЕЛЬНО	13
0<>	*	W->F	F "ИСТИНА", ЕСЛИ W НЕ НУЛЬ	19
0!	*	A->	ЗАСЛАТЬ НУЛЬ ПО АДРЕСУ A	11
0=		W->F	F "ИСТИНА", ЕСЛИ W РАВНО НУЛЮ	13
1+		W1->W2	УВЕЛИЧИТЬ W1 НА 1	17
1+!	*	A->	УВЕЛИЧИТЬ НА 1 ЗНАЧЕНИЕ ПО АДРЕСУ A	17
1-		W1->W2	УМЕНЬШИТЬ W1 НА 1	17
2+		W1->W2	УВЕЛИЧИТЬ W1 НА 2	17
2!		WD,A->	ЗАСЛАТЬ ДВОЙНОЕ WD ПО АДРЕСУ A	20
2*	*	W1->W2	АРИФМЕТИЧЕСКИЙ СДВИГ ВЛЕВО НА 1	20
2-		W1->W2	УМЕНЬШИТЬ W1 НА 2	17
2/		W1->W2	АРИФМЕТИЧЕСКИЙ СДВИГ ВПРАВО НА 1	20
2@		A->WD	ДАТЬ ДВОЙНОЕ ЗНАЧЕНИЕ ПО АДРЕСУ A	20
2CONSTANT		WD->	ОПРЕДЕЛИТЬ СЛЕДУЮЩЕЕ СЛОВО КАК КОНСТАНТУ СО ЗНАЧЕНИЕМ WD	32
2DROP		WD->	СНЯТЬ ВЕРХНЕЕ ДВОЙНОЕ ЗНАЧЕНИЕ	12
2DUP		WD->WD,WD	ПРОДУБЛИРОВАТЬ ДВОЙНОЕ ЗНАЧЕНИЕ	12
2LIT	*К	->WD	ПОМЕСТИТЬ НА СТЕК СЛЕДУЮЩИЕ 2 КОДА	23
2LITERAL	*Н	WD-> /КОМПИЛЯЦИЯ/ ->WD /ИСПОЛНЕНИЕ/	СКОМПИЛИРОВАТЬ WD КАК ЛИТЕРАЛ	23
2OVER		WD1,WD2->WD1,WD2,WD1	ПРОДУБЛИРОВАТЬ ВТОРОЕ ДВОЙНОЕ СВЕРХУ	12
2POP	М		ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР СО СНЯТИЕМ ДВОЙНОГО ЗНАЧЕНИЯ С ВЕРШИНЫ СТЕКА	3
2POPPUT1	М		ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР СО СНЯТИЕМ ДВОЙНОГО ВЕРХНЕГО ЗНАЧЕНИЯ СО СТЕКА И ЗАМЕНОЙ ПРЕДЫДУЩЕГО ДВОЙНОГО НА 4-БАЙТНОЕ ЗНАЧЕНИЕ ИЗ РЕГИСТРА RW1	3
2PUSHRW1	М		ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР С ПОМЕЩЕНИЕМ ДВОЙНОГО ЗНАЧЕНИЯ ИЗ RW1 НА ВЕРШИНУ СТЕКА	3
2PUTRW1	М		ВХОД В АДРЕСНЫЙ ИНТЕРПРЕТАТОР С ЗАМЕНОЙ ДВОЙНОГО ВЕРХНЕГО ЗНАЧЕНИЯ НА 4-БАЙТНОЕ ЗНАЧЕНИЕ ИЗ РЕГИСТРА RW1	3
2ROT		WD1,WD2,WD3->WD2,WD3,WD1	ПЕРЕСТАВИТЬ ТРИ ВЕРХНИХ ДВОЙНЫХ ПО ЧАСОВОЙ СТРЕЛКЕ	12
2SWAP		WD1,WD2->WD2,WD1	ОБМЕНЯТЬ МЕСТАМИ ДВА ВЕРХНИХ ДВОЙНЫХ ЗНАЧЕНИЯ	12
2VARIABLE		->	ОПРЕДЕЛИТЬ СЛЕДУЮЩЕЕ СЛОВО КАК ПЕРЕМЕННУЮ ДВОЙНОЙ ДЛИНЫ С НАЧАЛЬНЫМ ЗНАЧЕНИЕМ НУЛЬ	32
ФОРТ-СИСТЕМА	*	->	ТЕКСТОВЫЙ ИНТЕРПРЕТАТОР ФОРТ-СИСТЕМЫ	40

Экран номер 1

```
( 09.09.86      НАЧАЛО МОДЕЛИ ФОРТ-СИСТЕМЫ          )
DECIMAL     ( КОНСТАНТЫ ПЕРИОДА КОМПИЛЯЦИИ  )
128  CONSTANT  &IFLAG   ( ПРИЗНАК "IMMEDIATE")
32   CONSTANT  &SFLAG   ( ПРИЗНАК "SMUDGE")
31   CONSTANT  &LENG    ( МАСКА ДЛЯ ВЫСЕЧЕНИЯ ДЛИНЫ)
&SFLAG 256 * 64 + CONSTANT &DWORD  ( ФИКТИВНОЕ ИМЯ )
   (  НАЧАЛЬНОЕ ЯДРО С АДРЕСАЦИЕЙ ОТ РЕГИСТРА RFORTH)
START-CODE   *, RFORTH USING,  ( АДРЕСНЫЙ ИНТЕРПРЕТАТОР)
M:  NEXT    14 0 (, RI RFORTH LH,   RI RTWO AR,
M:  NEXT1   14 RMASK NR,   15 0 (, 14 RFORTH LH,
    15 RMASK NR,   15 RFORTH AR,   14 RTWO AR,   15 BR,
```

Экран комер 2

```
( 09.09.86  СИСТЕМНЫЕ ПЕРЕМЕННЫЕ И КОНСТАНТЫ )
CONST MSG      M: MSG#    0 H,    ( АДРЕС НАЧАЛА БУФЕРА MSG)
CONST FIRST    M: FIRST# 0 H,          ( АДРЕС НАЧАЛА ПУЛА)
CONST LIMIT    M: LIMIT# 0 H,           ( АДРЕС КОНЦА ПУЛА)
CONST TIB      M: TIB#    0 H,    ( АДРЕС НАЧАЛА БУФЕРА TIB)
  4 ALIGN
M: LENGMASK 255 &IFLAG - S>D F, ( БАЙТ ДЛИНЫ БЕЗ IMMEDIATE)
M: LENG1MSK 255 &IFLAG - &SFLAG - S>D F, ( БЕЗ IMMD И SMDG)
M: LENG2MSK &LENG S>D  F,     ( БАЙТ ДЛИНЫ С ЧИСТОЙ ДЛИНОЙ)
  8 ALIGN
M: TEMP  16 ALLOT                          ( РАБОЧАЯ ОБЛАСТЬ)
```

Экран номер 3

```
( 09.09.86 ДОПОЛНИТЕЛЬНЫЕ ВХОДЫ В АДРЕСНЫЙ ИНТЕРПРЕТАТОР)
M: DOES#       RI RPUSH,  RI 4 (, 15 LA,  RI RFORTH SR,
A: CREATE#     RW1 14 LR,               ( ПОМЕСТИТЬ PFA СТАТЬИ)
M: PUSHRW1     RSTACK RTWO SR,    ( ПОМЕСТИТЬ ЗНАЧЕНИЕ ИЗ RW1)
M: PUTRW1      RW1 PUT,    RNEXT BR,        ( ЗАМЕНИТЬ ВЕРХНЕЕ)
M: 2POP        RSTACK RTWO AR,            ( СНЯТЬ ДВА ВЕРХНИХ)
M: POP         RSTACK RTWO AR,   RNEXT BR,   ( СНЯТЬ ВЕРХНЕЕ)
M: POPPUT1     RSTACK RTWO AR,     ( СНЯТЬ ВЕРХНЕЕ И ЗАМЕНИТЬ)
               RW1  PUT,  RNEXT BR,        ( ЗНАЧЕНИЕМ ИЗ RW1)
M: 2PUSHRW1    RSTACK RTWO SR,     ( ПОЛОЖИТЬ ДВОЙНОЕ НА СТЕК)
M: PUSH2RW1    RSTACK RTWO SR, ( ЗАМЕНИТЬ ВЕРХНЕЕ НА ДВОЙНОЕ)
M: 2PUTRW1     RW1 TEMP ST,        ( ЗАМЕНИТЬ ДВОЙНОЕ ВЕРХНЕЕ)
               FIRST (, 4 ), TEMP MVC,    RNEXT BR,
M: 2POPPUT1    RSTACK RTWO AR,   RSTACK RTWO AR,    2PUTRW1 B,
```

Экран номер 4

```
( 09.09.86 ВСПОМОГАТЕЛЬНЫЕ ПОДПРОГРАММЫ: ВОЗВРАТ В РЕГ.14)
M:  LHRW12    RW1 SECOND LH,    RW2 PULL,    14 BR,
M:  LRW1      TEMP (, 4 ), FIRST MVC,    RW1 TEMP L,    14 BR,
M:  LRW12     TEMP (, 8 ), FIRST MVC,    RW1 TEMP 4 +(, L,
              RW2 TEMP L,    14 BR,
M:  GOTO      14 0 (, 0 14 LH,    NEXT1 B,
M:  IPUSH     RI PUSH,    RW2 RW2 SR,    RW2 0 (, RI RFORTH IC,
              RI 2 (, RI RW2 LA,    14 BR,
CODE EXIT
M: EXIT#  RI RPOP,    RI RMASK NR,    RNEXT BR,    END-CODE
M:  ERCONDG    14   GOTO BAL,    ] ABORTG [
```

Экран номер 5

```
( 09.09.86  BRANCH  ?BRANCH   (LOOP/  (+LOOP/ )
CODE  BRANCH  M: BRANCH#
   RI 0 (, RI RFORTH LH,    RI RMASK NR,      RNEXT BR,
CODE ?BRANCH  RW1 POP,   RW1 RW1 LTR,   BRANCH# BZ,
              RI RTWO AR,   RNEXT BR,
CODE  (LOOP)  RW1 1 LA,   1 =F B,
CODE (+LOOP)  RW1 POP,
1 =H    O RFIRST LH,    O RSECOND SH,   O RMASK NR,
        O RW1 AR,       RW1 RFIRST AH,  RW1 RFIRST STH,
        O RMASK CLR,    BRANCH# BNH,    RRET 6 (, O RRET LA,
        RI RTWO AR,   RNEXT BR,   END-CODE
```

Экран номер 6

```
( 09.09.86 KEY CR EMIT TYPE (EXPECT/ RBLK WBLK )
(  СЛЕДУЮЩИЕ ОПРЕДЕЛЕНИЯ ДАЮТ ТОЛЬКО ИМЕНА ПРОЦЕДУР)
CODE KEY   ( ->C  ВВЕСТИ ЛИТЕРУ С ТЕРМИНАЛА)       END-CODE
CODE CR    ( ->  ВЫВЕСТИ ПЕРЕВОД СТРОКИ )          END-CODE
CODE EMIT   ( C->   ВЫВЕСТИ ЛИТЕРУ С КОДОМ С НА ТЕРМИНАЛ)
                                                   END-CODE
CODE TYPE  ( A,N->  ВЫВЕСТИ НА ТЕРМИНАЛ N ЛИТЕР ПО АДРЕСУ А)
                                                   END-CODE
CODE (EXPECT) ( A,N1->A,N2  ВВЕСТИ С ТЕРМИНАЛА НЕ БОЛЕЕ
         N1 ЛИТЕР /ДО ПЕРЕВОДА СТРОКИ/ В БУФЕР ПО АДРЕСУ А;
         N2 - ФАКТИЧЕСКОЕ ЧИСЛО ВВЕДЕННЫХ ЛИТЕР) END-CODE
CODE RBLK  ( A,N-> ПРОЧИТАТЬ ЭКРАН N В БУФЕР А) END-CODE
CODE WBLK  ( A,N->  ЗАПИСАТЬ ЭКРАН N ИЗ БУФЕРА А ) END-CODE
```

Экран номер 7

```
( 09.09.86 КОНСТАНТЫ И СИСТЕМНЫЕ ПЕРЕМЕННЫЕ       )
64    CONSTANT BL     ( КОД ПРОБЕЛА)
1024  CONSTANT B/BUF ( ДЛИНА БУФЕРА ДЛЯ ЭКРАНА)
&LENG CONSTANT WIDTH    ( МАКСИМАЛЬНАЯ ДЛИНА СЛОВА )
0     CONSTANT 0        ( ЧИСЛО НОЛЬ)
VARIABLE USE   ( СЛЕДУЮЩИЙ БУФЕР В ПУЛЕ)
VARIABLE PREV  ( ТЕКУЩИЙ БУФЕР В ПУЛЕ)
VARIABLE  S0                      ( АДРЕС ДНА СТЕКА ДАННЫХ)
VARIABLE  R0                    ( АДРЕС ДНА СТЕКА ВОЗВРАТОВ)
VARIABLE  FENCE               ( ГРАНИЦА ЗАЩИТЫ ОТ "FORGET")
VARIABLE  CONTEXT         ( ТЕКУЩИЙ СПИСОК - НАЧАЛО ПОИСКА)
VARIABLE  CURRENT        ( ТЕКУЩИЙ СПИСОК - КУДА ДОБАВЛЯЕМ)
```

Экран номер 8

```
( 09.09.86   СИСТЕМНЫЕ ПЕРЕМЕННЫЕ - ОКОНЧАНИЕ)
VARIABLE  OFFSET                ( ДОБАВКА К НОМЕРУ ЭКРАНА)
VARIABLE  BASE              ( ОСНОВАНИЕ СИСТЕМЫ СЧИСЛЕНИЯ)
VARIABLE  STATE   ( СОСТОЯНИЕ ТЕКСТОВОГО ИНТЕРПРЕТАТОРА)
VARIABLE  DPL          ( ПОЗИЦИЯ ДЕСЯТИЧНОЙ ТОЧКИ В ЧИСЛЕ)
VARIABLE  CSP     ( ДЛЯ КОНТРОЛЬНОГО ХРАНЕНИЯ УКАЗАТЕЛЯ)
VARIABLE  HLD             ( УКАЗАТЕЛЬ ВЕРШИНЫ БУФЕРА "PAD")
VARIABLE  BLK             ( НОМЕР ВХОДНОГО ЭКРАНА ИЛИ НОЛЬ)
VARIABLE  >IN        ( ПОЗИЦИЯ ОЧЕРЕДНОЙ ЛИТЕРЫ НА ВХОДЕ)
VARIABLE  SPAN      ( ЧИСЛО ЛИТЕР, ВВЕДЕННЫХ ПО "EXPECT")
VARIABLE  #TIB   ( ЧИСЛО ЛИТЕР, ВВЕДЕННЫХ В БУФЕР "TIB")
VARIABLE  SCR    ( НОМЕР ЭКРАНА, РАСПЕЧАТАННОГО В "LIST")
```

Экран номер 9

```
( 31.03.86   DUP ?DUP DROP SWAP OVER >R R> R@ RDROP ROT )
CODE DUP    ( W->W,W) RW1 PULL,   PUSHRW1 B,   END-CODE
: ?DUP      ( W->W,W;  0->0 )   DUP IF DUP THEN ;
CODE DROP   ( W-> )  RSTACK RTWO AR,   RNEXT BR,   END-CODE
CODE SWAP   ( W1,W2->W2,W1)
   14 LHRW12 BAL,   RW2 SECOND STH,   PUTRW1 B,   END-CODE
CODE OVER ( W1,W2->W1,W2,W1) RW1 SECOND LH, PUSHRW1 B, END-CODE
CODE >R ( W-> )  RW1 POP,   RW1 RPUSH,   RNEXT BR,   END-CODE
CODE R>     ( ->W)     RW1 RPOP,   PUSHRW1 B,  END-CODE
CODE R@     ( ->W)     RW1 RPULL,   PUSHRW1 B,   END-CODE
CODE RDROP ( -> )     RRET RTWO AR,   RNEXT BR,   END-CODE
: ROT    ( N1,N2,N3->N2,N3,N1 )    >R SWAP R> SWAP ;
```

Экран номер 10

```
( 31.03.86        HERE ALLOT ALIGN ALIGNH DP! )
CODE HERE  ( ->A )  RW1 RD LR,   PUSHRW1 B,   END-CODE
CODE ALLOT  ( N-> )  RD FIRST AH,   POP B,   END-CODE
CODE ALIGN  ( N-> )  RW1 0 (, RD RFORTH LA,
    0 (, RW1 0 MVI,   1 (, 7 RW1 ), 0 (, RW1 MVC,
    RW1 PULL,   RW2 RW1 LCR,   RD RW1 AR,   RD 0 BCTR,
    RD RW2 NR,   POP B,   END-CODE
: ALIGNH   ( -> )  2 ALIGN ;
CODE DP!  ( A->  )  RD PULL,   RD RMASK NR,  POP B,
          END-CODE
```

Экран **номер** И

```
( 31.03.86       !  0!  @  C!  C@  ,  C,  EXECUTE  )
CODE !  ( W,A->    ЗАСЛАТЬ W ПО АДРЕСУ A)  14 LHRW12 BAL,
  RW2 RMASK NR,   RW1 0 (, RW2 RFORTH STH,   2POP B,  END-CODE
: 0!  ( A-> )   0 SWAP ! ;
CODE @   ( A->W  РАЗЫМЕНОВАТЬ A)   RW2 PULL,  RW2 RMASK NR,
  RW1 0 (, RW2 RFORTH LH,  PUTRW1 B,   END-CODE
CODE C@  ( A->C)  RW2 PULL,  RW2 RMASK NR,   RW1 RW1 SR,
  RW1 0 (, RW2 RFORTH IC,   PUTRW1 B,   END-CODE
CODE C!  ( C,A->   )      14 LHRW12 BAL,  RW2 RMASK NR,
   RW1 0 (, RW2 RFORTH STC,    2POP B,   END-CODE
: ,  ( W-> )  HERE 2 ALLOT ! ;
: C, ( C-> )  HERE 1 ALLOT C! ;
CODE EXECUTE  ( CFA-> )   14 POP,   NEXT1 B,   END-CODE
```

Экран номер 12

```
( 31.03.86   ROLL  PICK  2DUP  2DROP  2SWAP  2OVER  2ROT )
CODE ROLL  ( WN,WN-1,...,W0,+N->WN-1,...,W0,WN)
   RW2 PULL,   RW2 RW2 AR,   ERCOND8 BM,  RW1 SECOND (, RW2 LH,
   BEGIN, 0 FIRST (, RW2 LH,   0 SECOND (, RW2 STH,
     RW2 RTWO SR,   ?NP UNTIL,   POPPUT1 B,  END-CODE
CODE PICK  ( WN,...,W0,+N->WN,...,W0,WN)
   RW2 PULL,   RW2 RW2 AR,   ERCOND8 BM,
   RW1 2 (, RW2 RSTACK LH,   PUTRW1 B,   END-CODE
: 2DUP  ( WD->WD,WD)  OVER OVER ;
: 2DROP ( WD->)   DROP DROP ;
: 2SWAP ( WD1,WD2->WD2,WD1)  3 ROLL 3 ROLL ;
: 2OVER ( WD1,WD2->WD1,WD2,WD1)  3 PICK 3 PICK ;
: 2ROT  ( WD1,WD2,WD3->WD2,WD3,WD1) 5 ROLL 5 ROLL ;
```

Экран **номер** 13

```
( 31.03.86      AND  OR  XOR  NOT  0=  0<      )
CODE AND    ( W1,W2->W3)
  14 LHRW12 BAL,   RW1 RW2 NR,   POPPUT1 B,   END-CODE
CODE OR     ( W1,W2->W3)
  14 LHRW12 BAL,   RW1 RW2 OR,   POPPUT1 B,   END-CODE
CODE XOR    ( W1,W2->W3)
  14 LHRW12 BAL,   RW1 RW2 XR,   POPPUT1 B,   END-CODE
: NOT    ( W1->W2 )  -1 XOR ;
CODE 0=    ( W->F)  RW1 RW1 SR,   RW2 PULL,  RW2 RW2 LTR,
     PUTRW1 BNZ,   RW1 0 BCTR,   PUTRW1 B,   END-CODE
CODE 0<  ( N->F)  RW1 RW1 SR,   RW2 PULL,   RW2 RW2 LTR,
     PUTRW1 BNM,  RW1 0 BCTR,   PUTRW1 B,   END-CODE
```

Экран номер 14

```
( 31.03.86       S>D  DABS  DNEGATE  D+  D-  DU< )
CODE S>D  ( N->D ) RW1 PULL,   PUSH2RW1 B,  END-CODE
CODE DABS ( D1--D2)14 LRW1 BAL, RW1 RW1 LPR, 2PUTRW1 B, END-CODE
 CODE DNEGATE  ( WD1->WD2)
   14 LRW1 BAL,   RW1 RW1 LCR,   2PUTRW1 B,   END-CODE
CODE D+      ( WD1,WD2->WD3)
   14 LRW12 BAL,   RW1 RW2 AR,   2POPPUT1 B,   END-CODE
CODE D- .    ( WD1,WD2->WD3)
   14 LRW12 BAL,   RW1 RW2 SR,   2POPPUT1 B,   END-CODE
CODE DU< ( UD1,UD2->F) 14 LRW12 BAL,   0 0 SR,   RW1 RW2 CLR,
      ?L IF, 0 0 BCTR, THEN,  RSTACK 6 (, 0 RSTACK LA,
     0 PUT, RNEXT BR, END-CODE
```

Экран номер **15**

```
( 31.03.86        D/MOD  D/  DMOD  D0=  D=  D0<  D<  D2/ )
CODE D/MOD  ( D1,D2->D3,D4) 14 LRW12 BAL, 1 RW1 LR,  0 RW2 LR,
   RW1 32 SRDA,  RW1 0 DR,  1 0 XR,  1 1 LTR,  ?M IF,  RW1 0 AR,
   RW2 0 BCTR, THEN,  RW1 TEMP ST,  FIRST 4 +(, 4 ), TEMP MVC,
   RW1 RW2 LR,  2PUTRW1 B,   END-CODE
: D/    ( D1,D2->D3)  D/MOD 2SWAP 2DROP ;
: DMOD ( D1,D2->D3)  D/MOD 2DROP ;
: D0=  ( WD->F)   OR 0= ;
: D=   ( WD1,WD2->F). D- D0= ;
: D0<  ( D->F)    SWAP DROP 0< ;
: D<   ( D1,D2->F)   D- D0< ;
CODE D2/ ( D1->D2)  14 LRW1 BAL, RW1 1 SRA, PUTRW1 B, END-CODE
```

Экран камер **16**

```
( 09.09.86        UM/MOD  U<  M/MOD DMAX  DMIN )
CODE UM* ( U1,U2->UD)      14 LHRW12 BAL,    RW1 RMASK NR,
  RW2 RMASK NR, RW1 RW1 MR,  RW1 RW2 LR, 2PUTRW1 B, END-CODE
CODE UM/MOD  ( UD,U1->U2,U3) 1 POP,  1 RMASK NR,  14 LRW1 BAL,
  RSTACK RTWO SR,  RW2 RW1 LR,  RW1 RW1 SR,  RW1 1 DR,
  RW1 FIRST 4 +(, STH,  RW1 RW2 LR,  POPPUT1 B,  END-CODE
CODE U<    ( U1,U2->F)     RW1 RW1 SR,  RW2 PULL,  RW2 RMASK NR,
  0 SECOND LH,   0 RMASK NR,   0 RW2 CR,  POPPUT1 BNL,
  RW1 0 BCTR, ( РЕЗУЛЬТАТ "ИСТИНА")   POPPUT1 B, END-CODE
: M/MOD ( UD1,U2->U3,UD4) >R 0 R@ UM/MOD R> SWAP >R UM/MOD R> ;
: DMAX  ( D1,D2->D3)  2OVER 2OVER  D< IF 2SWAP THEN  2DROP ;
: DMIN ( D1,D2->D3)  2OVER 2OVER  D< NOT IF 2SWAP THEN 2DROP ;
```

Экран номер 17

```
( 31.03.86   NEGATE  ABS  +  -  1+  1-  2+  2-  +!  1+!  )
CODE NEGATE ( W1->W2) RW1 PULL, RW1 RW1 LCR, PUTRW1 B, END-CODE
: ABS  ( N1->+N2)  S>D  DABS  DROP ;
CODE +    14 LHRW12 BAL,   RW1 RW2 AR,   POPPUT1 B,   END-CODE
: -    ( W1,W2->W3)   NEGATE + ;
: 1+   ( W1->W2)    1 + ;
: 1-   ( W1->W2)    -1 + ;
: 2+   ( W1->W2)    2 + ;
: 2-   ( W1->W2)    2 - ;
CODE +!    ( W,A->)   14 LHRW12 BAL,   RW2 RMASK NR,   RW1 0 (,
     RW2 RFORTH AH, RW1 0 (, RW2 RFORTH STH, 2POP B, END-CODE
: 1+! ( A->)       1 SWAP +! ;
```

Экран номер 19

```
( 03.10.86      M*  M/  *  /MOD  /  MOD  */MOD  */)
CODE M* ( N1,N2->D) RW1 SECOND LH,  RW1 FIRST MH,  2PUTRW1 B,
CODE M/ ( D,N1->N2,N3) 1 POP,  14 LRW1 BAL,  RSTACK RTWO SR,
   RW1 32 SRDA,   0 RW1 LR,   RW1 1 DR,  0 1 XR,  0 0 LTR,
    ?M IF, RW1 1 AR, RW2 0 BCTR, THEN,  RW1 FIRST 4 +(, STH,
   RW1 RW2 LR,   ( ЧАСТНОЕ)  POPPUT1 B,   END-CODE
: *  ( N1,N2->N3)    M* DROP ;
: /MOD  ( N1,N2->N3,N4)  >R S>D R> M/ ;
: /   ( N1,N2->N3)    /MOD SWAP DROP ;
: MOD ( N1,N2->N3)  /MOD DROP ;
: */MOD ( N1,N2,N3->N4,N5)  >R M* R> M/ ;
: */    ( N1,N2,N3->N4)  */MOD SWAP DROP ;
```

Экран **номер 19**

```
( 31.03.86   СРАВНЕНИЯ И РАЗРЕШЕНИЯ В ШИТОМ КОДЕ)
: 0<>  ( N->F)    0= NOT ;
: =         ( W1,W2->F)     - 0= ;
: <>        ( W1,W2->F)     - 0<> ;
: <         ( N1,N2->F)     - 0<  ;

: >MARK       ( ->A )  HERE 0 , ;
: >RESOLVE   ( A->) HERE SWAP ! ;
: <MARK       ( ->A)    HERE ;
: <RESOLVE   ( A->) , ;
```

Экран номер 20

```
( 31.03.86      SP@  SP!  RP@  RP!  2/  2*  2@  2!  DEPTH )
CODE SP@  ( ->A)  RW1 RSTACK LR,   RW1 RFORTH SR,   PUSHRW1 B,
CODE SP!     ( A-> )   RSTACK PULL,   RSTACK RMASK NR,
     RSTACK RFORTH AR,   RNEXT BR,   END-CODE
CODE RP@ ( ->A) RW1 RRET LR, RW1 RFORTH SR, PUSHRW1 B, END-CODE
CODE RP! ( A->) RRET POP,    RRET RMASK NR,
    RRET RFORTH AR,   RNEXT BR,   END-CODE
CODE 2/ ( W1->W2 ) RW1 PULL,  RW1 1 SRA,  PUTRW1 B, END-CODE
: 2*   ( W1->W2 )   DUP + ;
: 2@       ( A->WD)  DUP 2+ @ SWAP @ ;
: 2!       ( WD,A->) DUP >R ! R> 2+ ! ;
: DEPTH    ( ->+N)    SP@ S0 @ SWAP - 2/ ;
```

Экран номер 21

```
( 31.03.86        CMOVE   CMOVE> )
CODE CMOVE  ( A1,A2,U->) 14 LHRW12 BAL, RW2 RMASK NR, 2 =F BZ,
   RW1 RMASK NR,  RW1 RFORTH AR,  1 FIRST 4 +(, LH,  1 RMASK NR,
   1 RFORTH AR, 0 256 LA,  1 =F B,  BEGIN,
     0 (, 256 RW1 ), 0 (, 1 MVC,  RW1 0 AR,  1 0 AR,
1 =H RW2 0 SR,  ?M UNTIL,  RW2 0 BCTR,  RW2 0 AR,
     ?NM IF, RW2 3 =F EX, THEN,
2 =H RSTACK  6 (, 0 RSTACK LA,  RNEXT BR,
3 =H 0 (, 1 RW1 ), 0 (, 1 MVC,   END-CODE
CODE CMOVE>  ( A1,A2,U->) 14 LHRW12 BAL, RW2 RMASK NR, 1 =F BZ,
   RW1 RMASK NR,   RW1 RFORTH AR,   RW1 0 BCTR,
   1 FIRST 4 +(, LH,    1 RMASK NR,   1 RFORTH AR,   1 0 BCTR,
   DO, 0 0 (, 1 RW2 IC,  0 0 (, RW1 RW2 STC,  RW2 LOOPBCT,
1 =H   RSTACK 6 (, 0 RSTACK LA,   RNEXT BR,  END-CODE
```

Экран номер 22

```
( 31.03.86   FILL ERASE BLANK COMPILE [ ] MIN MAX HEX DECIMAL)
: FILL     ( A,U,C->) SWAP ?DUP IF >R OVER C!
           DUP 1+ R> 1- CMOVE EXIT THEN 2DROP ;
: ERASE    ( A,U-> )  0 FILL ;
: BLANK    ( A,U-> )  BL FILL ;
: COMPILE  ( -> )  R> DUP 2+ >R @ , ;
: [        ( -> )  STATE 0! ;  IMMEDIATE
: ]        ( -> ) -1 STATE ! ;
CODE MIN    ( N1,N2->N3 )   14 LHRW12 BAL,   RW1 RW2 CR,
       POP BNH,   RW1 RW2 LR,   POPPUT1 B,   END-CODE
CODE MAX    ( N1,N2->N3 )   14 LHRW12 BAL,   RW1 RW2 CR,
       POP BNL,   RW1 RW2 LR,   POPPUT1 B,   END-CODE
: HEX ( ->)  16 BASE ! ;
: DECIMAL ( ->) 10 BASE ! ;
```

Экран номер 23

```
( 31.03.86     LIT  2LIT  LITERAL  2LITERAL  SPACE  SPACES )
CODE LIT    ( ->W    )       RW1 0 (, RI RFORTH LH,
   RI RTWO AR,   PUSHRW1 B,    END-CODE
CODE 2LIT   ( ->WD )   RW1 4 LA,   RSTACK RW1 SR,
   RW2 0 (, RI RFORTH LA,   FIRST (, 4 ), 0 (, RW2 MVC,
   RI RW1 AR,   RNEXT BR,   END-CODE
: LITERAL   ( W->) STATE @ IF COMPILE LIT , THEN ; IMMEDIATE
: 2LITERAL ( WD->) STATE @ IF COMPILE 2LIT , , THEN ; IMMEDIATE
: SPACE ( ->)        BL EMIT ;
: SPACES ( +N->) 0 OVER < IF 0 DO SPACE LOOP EXIT THEN  DROP ;
```

Экран номер 24

```
( 09.09.86 ЦИКЛЫ СО СЧЕТЧИКОМ:     (DO/ I I' J   LEAVE)
CODE (DO) ( U1,U2-> )  14 LHRW12 BAL,
1 =H        1 0 (, RI RFORTH LH,   1 RPUSH,   RI RTWO AR,
            RW1 RPUSH,   RW2 RPUSH,   2POP B,  END-CODE
CODE I      ( ->U   ТЕКУЩЕЕ ЗНАЧЕНИЕ СЧЕТЧИКА ЦИКЛА)
            RW1 RPULL,   PUSHRW1 B,   END-CODE
CODE I'     ( ->U   ВЕРХНЯЯ ГРАНИЦА ЦИКЛА)
            RW1 RSECOND LH,   PUSHRW1 B,   END-CODE
CODE J      ( ->U   ТЕКУЩЕЕ ЗНАЧЕНИЕ СЧЕТЧИКА 2-ГО ЦИКЛА)
            RW1 RFIRST 6 +(, LH,   PUSHRW1 B,   END-CODE
CODE LEAVE ( ->)   RI RFIRST 4 +(, LH,   RI RMASK NR,
            RRET 6 (, 0 RRET LA,   RNEXT BR,   END-CODE
```

Экран номер 25

```
( 31.03.86    +BUF  BUFFER  BLOCK  EMPTY-BUFFERS  UPDATE   )
: +BUF   ( A1->A2,F ПЕРЕЙТИ К СЛЕДУЮЩЕМУ БУФЕРУ В ПУЛЕ)
 B/BUF 4 + + DUP LIMIT = IF DROP FIRST THEN DUP PREV @ - ;
: BUFFER ( +N->A)  OFFSET @ + USE @ DUP >R
    ( ИЩЕМ СВОБОДНЫЙ БУФЕР)     BEGIN +BUF UNTIL USE !
    R@ @ 0< IF ( УСТАНОВЛЕН ПРИЗНАК "UPDATE")
      R@ 2+ R@ @ 32767 AND WBLK THEN   R@ ! R@ PREV ! R> 2+ ;
: BLOCK ( +N->A)  OFFSET @ + >R PREV @ DUP @ R@ - DUP + IF
    BEGIN +BUF 0= IF DROP R@ OFFSET @ - BUFFER DUP R@ RBLK
                 2- THEN    DUP @ R@ - DUP + 0=
    UNTIL DUP PREV ! THEN RDROP 2+ ;
: EMPTY-BUFFERS_ ( -> )   FIRST LIMIT OVER - ERASE ;
: UPDATE ( -> )   PREV @ @ 32768 OR PREV @ ! ;
```

Экран **номер** 26

```
( 31.03.86     SAVE-BUFFERS  FLUSH  )
: SAVE-BUFFERS  ( ->)
    LIMIT FIRST DO I @ 32768 AND
                  IF    I @ 32767 AND DUP I !
                           I 2+ SWAP WBLK
                  THEN
     B/BUF 4 + +LOOP
;
: FLUSH  ( ->)  SAVE-BUFFERS   EMPTY-BUFFERS ;
```

Экран номер 27

```
( 31.03.86    ENCLOSE  WORD )
CODE ENCLOSE ( A,C->A,N1,N2,N3) 14 LHRW12 BAL, RW1 RMASK NR,
     RW1 RFORTH AR,  14 14 SR,   0 0 SR,
  BEGIN,   0 0 (, 14 RW1 IC,   0 0 LTR,   2 =F BZ,
           14 1 (, 0 14 LA,  0 RW2 CR, ?NE UNTIL,  14 0 BCTR,
2 =H   14 PUT,
  BEGIN,   1 14 LR,   0 0 (, 1 RW1 IC,   0 0 LTR,
           2 =F BZ,   14 1 (, 0 14 LA, 0 RW2 CR,   ?E UNTIL,
2 =H   1 PUSH,   RW1 14 LR,   PUSHRW1 B,   END-CODE
: WORD   ( C->T )   BLK @ IF BLK @ BLOCK ELSE TIB THEN
      >IN @ + SWAP ENCLOSE >IN +!
      HERE >R OVER - >R + ALIGNH HERE 1+ R@ CMOVE
      HERE R> 1+ ALLOT ALIGNH HERE OVER - 2- OVER C!  R> DP! ;
```

Экран номер 28

```
( 31.03.86 LIT" COUNT ," " ". (."/ ." C" ( .( QUIT ABORT )
CODE LIT"   ( ->T )  14 IPUSH BAL,   RNEXT BR,   END-CODE
: COUNT ( T->A,N)  DUP 1+ SWAP C@ 2DUP + C@ IF 1+ THEN ;
: ," ( -> ) C" " WORD C@ 2+ ALLOT ALIGNH ;
: "  ( ->T)  ?COMP  COMPILE LIT"  ," ; IMMEDIATE
: ".  ( T-> )   COUNT TYPE ;
CODE (.")  ( ->) 14 IPUSH BAL,  14 GOTO BAL, ] ". [
: ."  ( -> )  ?COMP  COMPILE (.")  ," ;  IMMEDIATE
: C" ( ->C)  BL WORD 1+ C@  [COMPILE] LITERAL ; IMMEDIATE
: (  ( ->)  C" )  WORD DROP ; IMMEDIATE
: .( ( ->)  C" ) WORD COUNT TYPE ; IMMEDIATE
: QUIT  ( ->) [COMPILE] [  S0 @ SP!  R0 @ RP!  ФОРТ-СИСТЕМА ;
```

Экран номер 29

```
( 31.03.86 ПРОВЕРКИ И СИГНАЛИЗАЦИИ ОБ ОШИБКАХ )
: ?ABORT ( F,T->)  SWAP IF COUNT CR TYPE ABORT THEN DROP ;
CODE (A") ( F->) 14 IPUSH BAL, 14 GOTO BAL, ] ?ABORT [ END-CODE
: ABORT"  ( F->)  COMPILE (A") ," ;   IMMEDIATE
: ABORT8  ( ->)  -1 ABORT" НЕПРАВИЛЬНОЕ ЗНАЧЕНИЕ В СТЕКЕ" ;
: !CSP  ( ->)   SP@ CSP ! ;
: ?CSP  ( ->)  SP@ CSP @ - ABORT" СБИЛСЯ УКАЗАТЕЛЬ СТЕКА" ;
: ?PAIRS  ( N1,N2-> )  - ABORT" НЕПАРНЫЕ СКОБКИ" ;
CODE ?+ ( N->N ) FIRST 128 TM, RNEXT BZR, ERCOND8 B, END-CODE
: ?COMP ( ->) STATE @ NOT ABORT" ТРЕБУЕТСЯ РЕЖИМ КОМПИЛЯЦИИ" ;
: BADWORD ( T->)   CR  ".  ."   ?"  ABORT ;
```

Экран номер 30

```
( 31.03.86  >BODY BODY> >LINK LINK> L>NAME N>LINK >NAME NAME> )
: >BODY   ( CFA->PFA) 2+ ;
: BODY>   ( PFA->CFA) 2- ;
: >LINK   ( CFA->LFA) 2- ;
: LINK>   ( LFA->CFA) 2+ ;
CODE L>NAME  ( LFA->NFA) RW2 PULL,   RW2 RMASK NR,  RW1 RW2 LR,
  14 &LENG LA, 1 1 SR,  DO, RW1 RTWO SR,  1 0 (, RW1 RFORTH IC,
  1 LENG1MSK N,    0 2 (, 1 RW1 LA,  0 RW2 CR,  PUTRW1 BE,
  14 LOOPBCT,  PUTRW1 B,   END-CODE
: N>LINK  ( NFA->LFA)  DUP C@ 31 AND + 2+ ;
: >NAME ( CFA-NFA)   >LINK L>NAME ;
: NAME> ( NFA->CFA) N>LINK LINK> ;
```

Экран номер 31

```
( 31.03.86    LATEST DEFINITIONS SMUDGE UNSMUDGE IMMEDIATE ID.)
: LATEST    ( ->NFA)   CURRENT @ @ ;          ( (;CODE/ RECURSE)
: DEFINITIONS ( ->) CONTEXT @ CURRENT ! ;
: SMUDGE  ( ->)    LATEST C@ [ &SFLAG ] LITERAL OR LATEST C! ;
: UNSMUDGE ( ->)   LATEST C@ [ 255 &SFLAG - ] LITERAL
                   AND LATEST C! ;
: IMMEDIATE ( ->)  LATEST C@ [ &IFLAG ] LITERAL OR LATEST C! ;
: ID.   ( NFA-> )  DUP 1+ SWAP C@ [ &LENG ] LITERAL AND
                   2DUP + C@ IF 1+ THEN TYPE SPACE ;
: (;CODE)  ( -> )   R>  LATEST  NAME>  ! ;
: RECURSE  ( ->)    LATEST NAME> , ;  IMMEDIATE
```

Экран номер 32

```
( 31.03.86   CONSTANT  VARIABLE  2CONSTANT  2VARIABLE  :  ; )
: ?LOADING ( ->) BLK @ 0= ABORT" НЕТ ОБРАБОТКИ ЭКРАНА" ;
: ?GAP ( N->) HERE + SP@ SWAP U< ABORT" ИСЧЕРПАНИЕ ПАМЯТИ" ;
: ?STACK ( ->) S0 @ SP@ U< ABORT" ИСЧЕРПАНИЕ СТЕКА" 10 ?GAP ;
: CONSTANT ( W-> ) CREATE , ;CODE
               RW1 0 (, 14 RFORTH LH,   PUSHRW1 B,  END-CODE
: VARIABLE  ( -> ) CREATE 0 , ;
: 2VARIABLE ( -> ) CREATE 0 , 0 , ;
: 2CONSTANT ( WD-> ) CREATE , , DOES> 2@ ;
: :  ( -> ) !CSP CREATE ] SMUDGE ;CODE
                  RI RPUSH,   RI 14 LR,  RNEXT BR,   END-CODE
: ;  ( -> ) ?CSP COMPILE EXIT UNSMUDGE  [COMPILE] [ ; IMMEDIATE
```

Экран номер 33

```
( 09.09.86    FORTH FORTH# FL# VOC-LINK VOCABULARY VOCABULARY#)
VOC  FORTH    &DWORD  H,                              ( FORTH-83)
A: FORTH#  LASTWORD   ( ВХОД В СПИСОК СЛОВАРНЫХ СТАТЕЙ)
A: FL#          0 H,     ( ПОЛЕ СВЯЗИ ДЛЯ СПИСКОВ СТАТЕЙ)
CREATE VOC-LINK  FL#     ( ВХОД В СПИСОК СПИСКОВ СТАТЕЙ)
: VOCABULARY ( -> )  CREATE [ &DWORD ] LITERAL ,
       LIT [ FORTH# ]
   CONTEXT @ - IF CONTEXT @ 2- ELSE 0 THEN ,
   HERE VOC-LINK @ , VOC-LINK !  DOES>
         [  THERE  4  -  :A: VOCABULARY#   ]
              2+ CONTEXT ! ;
: FORTH-83  ( ->)  FORTH DEFINITIONS  DECIMAL ;
```

Экран **номер** 34

```
( 31.03.86    (FIND/  )
CODE (FIND) ( -1,AN,...,A1,T->CFA,C,TF/FF )  RW2 POP,
 RW2 RMASK NR,   RW2 RFORTH AR,  ( ОБРАЗЕЦ)  0 0 SR,
 0 0 (, 0 RW2 IC,  0 LENG1MSK N, ( ДЛИНА)  1 1 SR, 1 0 BCTR,
BEGIN, RW1 PULL,  ( ВХОД В ОЧЕРЕДНОЙ СПИСОК СЛОВ) 2 =F B,
  BEGIN, RW1 RFORTH AR,    14 0 (, 0 RW1 IC,  14 LENGMASK N,
    14 0 CR, ?E IF, 14 4 =F EX,   3 =F BE,   THEN,
    14 LENG1MSK  N,   RW1 2 (, 14 RW1 LH,
2 =H   RW1 RMASK NR,   ?Z UNTIL,
 RSTACK RTWO AR,   1 FIRST CH,   ?E UNTIL,  PUTRW1 B,
BEGIN, RSTACK RTWO AR,  3 =H   1 FIRST CH,  ?E UNTIL,
 0 0 (, 0 RW1 IC,   RW1 RFORTH SR,   RW1 4 (, 14 RW1 LA,
 RW1 PUT,   0 PUSH,    RW1 1 LR,   PUSHRW1 B,
4 =H   1 (, 1 RW1 ), 1 (, RW2 CLC,     END-CODE
```

Экран номер 35

```
( 31.03.86       FIND  -FIND )
: FIND   ( T->A,N)
    DUP >R -1   LIT [ FORTH# ] @
    CURRENT @ @ 2DUP = IF DROP THEN
    CONTEXT @ @ 2DUP = IF DROP THEN
    R> (FIND)    DUP IF
  DROP ROT DROP [ &IFLAG ] LITERAL AND IF 1 ELSE -1 THEN
                   THEN
;
: -FIND ( ->A,N)   BL WORD FIND ;
```

Экран **номер** 3A

```
(  09.09.86         CREATE   DOES> )
: CREATE ( ->))   100 ?GAP
            ALIGNH -FIND SWAP DROP    IF
      HERE ID. ." УЖЕ ЕСТЬ  "         THEN
     HERE DUP C@ WIDTH AND 2+ ALLOT ALIGNH
     HERE OVER - 2- OVER C! LATEST , CURRENT @ !
     LIT [ CREATE# ]   , ;

: DOES>  ( -> )  COMPILE (;CODE)  2LIT
      [ DOES#  B, ] , , ;   IMMEDIATE
```

Экран **номер** 37

```
( 31.03.86     PAD  HOLD  ALPHA  <#  #>  #  #S  SIGN )
: PAD   ( ->A)    HERE 100 + ;
: HOLD ( C-> )   -1 HLD +!  HLD @ C! ;
CODE ALPHA  ( N->C)  RW2 FIRST LH,
     RW1 RW1 SR,   RW1 1 =F (, RW2 IC,  PUTRW1 B,
1 =H  C,' 0123456789ABCDEFGHIJKLMNOPQRSTUVWZYZ'
  END-CODE
: <#   ( -> )   PAD HLD ! ;
: #>   ( D->A,+N)   2DROP HLD @ PAD OVER - ;
: #    ( D1->D2)   BASE @ M/MOD ROT ALPHA HOLD ;
: #S   ( D->0,0)   BEGIN # 2DUP OR 0= UNTIL ;
: SIGN ( N->)     0< IF C" - HOLD THEN ;
```

Экран **номер** **38**

```
( 31.03.86    D.R  D.  .R .  H.  U.  U.R  ?  )
: D.R   ( D,+N-> )   ?+  >R  DUP >R DABS
     <#  #S  R>  SIGN  #>  R>  OVER  -  SPACES  TYPE ;
: D.   ( D-> )    0  D.R SPACE ;
: .R   ( N1,+N2->)   >R  S>D  R>  D.R ;
: .   ( N-> )   S>D  D.  ;
: H.   ( N->)   BASE @ SWAP 0 HEX <# # # # # #> TYPE SPACE
                BASE ! ;
: U.   ( U->)  0  D. ;
: U.R ( U,+N->)   >R 0 R> D.R ;
: ?    ( A->)   @ . ;
```

Экран **39**

```
( 31.03.86       DIGIT  CONVERT  NUMBER )
: DIGIT ( C,N1->N2,TF/FF)  0 ROT ROT 0
   DO I ALPHA OVER = IF 2DROP I -1 0 LEAVE THEN  LOOP DROP ;
: CONVERT   ( WD1,A1->WD2,A2)
    BEGIN 1+ DUP >R C@ BASE @ DIGIT WHILE
          SWAP BASE @ UM* DROP ROT BASE @ UM* D+
          DPL @ 1+ IF DPL 1+! THEN R>  REPEAT R> ;
: NUMBER   ( T->WD )
     0 0 ROT DUP >R COUNT OVER + OVER C@ C" - =
     DUP >R SWAP >R IF ELSE 1- THEN -1
      BEGIN DPL !  CONVERT  DUP R@ <  WHILE  DUP C@
      C" . <> IF RDROP RDROP R> BADWORD THEN 0
      REPEAT  DROP RDROP R>    IF DNEGATE THEN RDROP ;
```

Экран **номер** 40

```
( 31.03.86      EXPECT  QUERY  INTERPRET  ФОРТ-СИСТЕМА  X )
: EXPECT   ( A,+N-> )     DUP >R   (EXPECT)  DUP  SPAN !
     TYPE R> SPAN @ - IF SPACE THEN ;
: QUERY ( ->) TIB 80 EXPECT >IN 0! BLK 0! SPAN @ #TIB ! ;
: INTERPRET ( ->)    BEGIN -FIND ?DUP IF
    1+ IF EXECUTE ELSE  STATE @  IF , ELSE EXECUTE THEN  THEN
         ELSE NUMBER DPL @ 1+ IF [COMPILE] 2LITERAL
           ELSE DROP [COMPILE] LITERAL THEN THEN ?STACK AGAIN ;
: ФОРТ-СИСТЕМА  ( ->)  BEGIN QUERY INTERPRET AGAIN ;
CODE X ( ->)  ЗАБИТЬ-X   ( НУЛЕВОЙ КОД ВМЕСТО БУКВЫ "X")
    EXIT# B,   END-CODE                   IMMEDIATE
```

Экран номер 41

```
( 31.03.86 -TRAILING  '  [']  [COMPILE]  LOAD  THRU  ;S  -->  )
CODE -TRAILING ( A,N1- >A,N2)    14 LHRW12 BAL,   RW1 RMASK NR,
  RW1 RFORTH AR,  0 RW1 LR,  RW1 RW2 AR,  BEGIN,  RW1 0 CR,
  1 =F BNH, RW1 0 BCTR, 0 (, RW1 64 CLI, ?NE UNTIL, 0 0 BCTR,
1 =H  RW1 0 SR,   PUTRW1 B,   END-CODE
: ' ( ->CFA)   -FIND 0= IF BADWORD THEN ;
: [']  ( -> )  ?COMP ' [COMPILE] LITERAL ; IMMEDIATE
: [COMPILE] ( ->)  -FIND  IF , EXIT THEN BADWORD ; IMMEDIATE
: LOAD   ( N->    ИНТЕРПРЕТИРОВАТЬ БЛОК С НОМЕРОМ N)
   >IN @ >R BLK @ >R BLK ! >IN 0! INTERPRET R> BLK ! R> >IN ! ;
: THRU   ( N1,N2->  ИНТЕРПРЕТИРОВАТЬ БЛОКИ ОТ N1 ДО N2)
    1+ SWAP DO I LOAD LOOP ;
: ;S  ( ->)   ?LOADING  RDROP ; IMMEDIATE
: --> ( ->)   ?LOADING  >IN 0!  BLK 1+! ;  IMMEDIATE
```

Экран номер 42

```
(  09.09.86   DUMP   SNAPSTK  S.  R. )
: DUMP   ( A,U->    РАСПЕЧАТАТЬ U БАЙТОВ)     DUP IF
    BASE @ >R   HEX OVER + SWAP DO CR I <# C" * HOLD
    0 15 DO   DUP I + C@  HOLD   -1 +LOOP  C" * HOLD
    0 15 DO BL HOLD DUP I +  C@ 0 # # 2DROP -1 +LOOP  BL HOLD
BL HOLD 0 # # # # #> TYPE 16 +LOOP  R> BASE ! ELSE 2DROP THEN ;
: SNAPSTK      RDROP CR ". ." , ВСЕГО ЗНАЧЕНИЙ "
    2DUP SWAP - 2/ DUP . 0 SWAP < IF  ." (ВЕРШИНА СПРАВА)" CR
    2- DO I @ . -2 +LOOP ELSE 2DROP THEN ;
: S. ( ->)       SP@ S0 @ " СТЕК ДАННЫХ" SNAPSTK ;
: R. ( ->)   RP@ 2+ R0 @ " СТЕК ВОЗВРАТОВ" SNAPSTK ;
```

Экран номер 43

```
( 31.03.86    .VOC  (VOC/  VOCS    )
: .VOC    ( PFA+2-> )   2- BODY> >NAME ID. ;
: (VOC)   ( PFA1+2->PFA2,N)     @ 0
     BEGIN OVER DUP IF @ [ &DWORD ] LITERAL <> THEN
     WHILE 1+ ( СЧЕТЧИК СЛОВ) SWAP N>LINK @ SWAP REPEAT ;
: VOCS  ( -> )   -1 ['] FORTH >BODY 2+
   CURRENT @ ." СПИСОК CURRENT: " DUP .VOC    OVER @ OVER @ =
   IF DROP THEN  CONTEXT @ ."  СПИСОК CONTEXT: " DUP .VOC
        OVER @ OVER @ = IF DROP THEN
   CR ." СТАНДАРТНЫЙ ПОРЯДОК ПОИСКА: "
   BEGIN 2- BEGIN 2+ DUP .VOC (VOC) DROP DUP 0= UNTIL
         DROP DUP -1 = UNTIL DROP
   CR ." НАЛИЧНЫЕ СПИСКИ СЛОВ: " VOC-LINK @
   BEGIN DUP 2- .VOC @ DUP 0= UNTIL DROP ;
```

Экран номер 44

```
( 31.03.86    WORDS  )
: WORDS ( -> )
   ." СПИСОК " CONTEXT @ DUP .VOC  DUP (VOC)
   ."  ВСЕГО СЛОВ - " . ."  СЛЕДУЮЩИЙ СПИСОК - "
   ?DUP IF 2+ .VOC THEN
  CR @  BEGIN DUP DUP IF @ [ &DWORD ] LITERAL <> THEN
        WHILE DUP C@ [ &SFLAG ] LITERAL AND 0=
              IF DUP ID. SPACE THEN
              N>LINK @ REPEAT DROP ;
```

Экран номер 43

```
( 31.03.86   (FORGET/  FORGET  REMEMBER FORGETO       )
: (FORGET)  ( A-> ИСКЛЮЧИТЬ ВСЕ СЛОВА ВЫШЕ АДРЕСА А)
   DUP FENCE @ U< ABORT" ЗАЩИТА ПО FENCE"
   >R VOC-LINK @
   BEGIN R@ OVER U< WHILE
         FORTH DEFINITIONS
         @ DUP VOC-LINK !
   REPEAT      ( ДОШЛИ ДО СПИСКА, ГДЕ ЕСТЬ ЭТО СЛОВО)
   BEGIN DUP 4 -
         BEGIN N>LINK @ DUP R@ U< UNTIL
         OVER 2- ! @ ?DUP 0=   UNTIL  R> DP! ;
: FORGET  ( ->)  ' >NAME (FORGET) ;
: REMEMBER ( ->)  CREATE  DOES>  (FORGET) ;
```

Экран номер 46

```
( 31.09.86    (#SCR/  LIST  SCR?  INDEX  )
: (#SCR)  ( N->A,T  ПЕРЕВЕСТИ НОМЕР N ЭКРАНА В ТЕКСТОВОЕ ИМЯ)
    BASE @ >R DECIMAL  0 <# #S #>  R> BASE ! ;
: LIST   ( N->  РАСПЕЧАТАТЬ ЭКРАН N, ЗАПОМНИТЬ ЕГО В "SCR")
    DUP SCR ! CR ." ЭКРАН " DUP (#SCR) TYPE
    BLOCK 16 0 DO DUP I 64 * +
               CR I 3 .R SPACE 64 TYPE LOOP DROP ;
```

Экран номер 47

```
( 31.03.86   СТАНДАРТНЫЕ СТРУКТУРЫ УПРАВЛЕНИЯ )
: BEGIN     ?COMP <MARK 1 ;                     IMMEDIATE
: UNTIL     1 ?PAIRS  COMPILE ?BRANCH  <RESOLVE ; IMMEDIATE
: AGAIN     1 ?PAIRS  COMPILE  BRANCH  <RESOLVE ; IMMEDIATE
: IF        ?COMP  COMPILE ?BRANCH  >MARK  2 ;    IMMEDIATE
: THEN      2 ?PAIRS  >RESOLVE ;                 IMMEDIATE
: ELSE      2 ?PAIRS  COMPILE BRANCH  >MARK
            SWAP  >RESOLVE  2 ;                  IMMEDIATE
: WHILE     1 ?PAIRS  1  [COMPILE] IF ;          IMMEDIATE
: REPEAT    >R  >R  [COMPILE]  AGAIN
            R>  R>  [COMPILE]  THEN  ;           IMMEDIATE
: DO        ?COMP  COMPILE (DO)  >MARK <MARK  3 ; IMMEDIATE
: LOOP  3 ?PAIRS  COMPILE (LOOP) <RESOLVE >RESOLVE ; IMMEDIATE
: +LOOP 3 ?PAIRS COMPILE (+LOOP) <RESOLVE >RESOLVE ; IMMEDIATE
```

Приложение 2. Распространенные форт-системы

Фиг-форт. Система разработана в 1978 — 1980 гг. группой из 9 системных программистов в США — Группой по языку Форт *(FORTH Interest Group),* желавших сделать этот язык удобным средством программирования для персональных ЭВМ [28]. Система реализована для целого ряда ЭВМ с различной архитектурой. В нашей стране получила распространение на ЭВМ СМ-4 с операционными системами ОС РВ, РАФОС

и без операционной системы. Ее ядро, написанное на макроязыке ассемблера, занимает от 4 до 5 К байт и после загрузки в память ЭВМ позволяет вводить следующие определения уже непосредственно на языке Форт. Общий объем словаря — около 8 К (220 слов). В реализации применен косвенный шитый код. Имеются загружаемый ассемблер и текстовый редактор.

Форт-СМ. Система разработана в Ленинграде С. Б. Кацевым (ЛГУ) и И. А. Шендриковым (ЛИТМО) на основе стандарта "Форт-83". Используется с 1985 г. на ЭВМ СМ-3, СМ-4, ДВК, «Электроника-60», БК0010. Словарь включает около 350 слов, его общий объем — 10,5 К. В зависимости от генерации может работать под операционными системами ОС РВ, РАФОС или без операционной системы с перфоленточной загрузки. В реализации применен косвенный шитый код. Имеются связь с файловой системой соответствующей ОС, загружаемый структурный ассемблер с метками, строковый и экранный редакторы, целевая компиляция.

Форт-Тарту. Система разработана в ВЦ Тартуского государственного университета Р. В. Вяйнасте и А. Э. Юуриком для операционных систем ОС РВ (используется с 1983 г.) и ЮНИКС (с 1985 г.). Ядро системы занимает 8 К байт и включает 270 слов. Система является расширением стандарта «Фиг-форт». В реализации применен прямой шитый код. Имеется встроенный ассемблер, в оттранслированном виде занимающий 1 К байт. Встроенный текстовый редактор для ОС РВ имеет только строковый режим, для ОС ЮНИКС реализован экранный вариант. Система используется в учебном процессе и как инструментальное средство для НИР [20].

Форт-К580. Система разработана в ЛГУ В. А. Кириллиным, А. А. Клубовичем и Н. Р. Ноздруновым для микропроцессора К580. Используется с 1983 г. Система легко переносится на любое оборудование на базе К580, в частности она успешно перенесена на ЕС-7970, СМ-1800 (под управлением ОС "Си-Пи-Эм"), К1-10, КТС ЛИУС, КУВТ «Ямаха», КУВТ «Корвет» и большое число мелкосерийных микроЭВМ. Включена в комплект заводской поставки ЕС-7970 в составе программного комплекса ЯНУС. Ядро системы занимает от 8 до 12 К байт и насчитывает около 300 слов. Система является расширением стандарта "Форт-83", включает все

стандартные расширения и ряд слов для взаимодействия с операционной системой и аппаратурой устройств, а также для разработки и отладки программ. Имеются отдельно загружаемые пакеты для работы с числами в формате с плавающей точкой, для диалогового обучения языку Форт, для целевой компиляции и построения конечного программного продукта, размещаемого в ПЗУ и (или) ОЗУ. В реализации применен прямой шитый код. Встроенный структурный ассемблер разрешает использование меток. Имеется два текстовых редактора — построчный и поэкранный.

Форт-ЕС. Система разработана в Ленинградском институте информатики и автоматизации АН СССР (ЛИИАН) С. Н. Барановым для ЕС ЭВМ [2] под управлением ОС ЕС и СВМ ЕС. Ядро системы занимает 13 К байт и насчитывает 350 слов. Система является расширением стандарта «Форт-83», включает все стандартные расширения и ряд слов для взаимодействия с операционной системой, для работы с четырехбайтными машинными адресами, для разработки и отладки программ. Имеются отдельно загружаемые пакеты для работы с числами в формате с плавающей точкой, для связи с файловой системой, для справочной подсистемы с диалоговым учебником по языку Форт. В реализации применен косвенный шитый код. Встроенный структурный ассемблер разрешает использование меток. Текстовый редактор работает в режимах построчного и поэкранного редактирования. Система имеет отдельно загружаемый пакет целевой компиляции и средства для построения независимого программного продукта

Форт-Искра-226. Эта развитая операционная система на базе языка Форт [6] разработана в Институте социально-экономических проблем (ИСЭП) АН СССР. Используется с 1986 г. Включена в комплект заводской поставки ЭВМ «Искра-226». Объем — около 32 К байт (свыше 400 слов). Система базируется на стандарте "Фиг-форт", расширенном рядом слов в соответствии с ее функциональным назначением. Среди них встроенный диспетчер, средства для параллельного выполнения директив, работа с файлами и базами данных, работа с адресным пространством до 128 К байт. Важным встроенным средством является программная реализация элементарных математических функций и

операций для работы с плавающей точкой. В реализации применен прямой шитый код. Встроенный экранный редактор имеет режим работы с окнами.

Форт-М6000. Система разработана В. Н. Патрышевым (Ленинград) для ЭВМ М6000 под управлением ДОС РВ, РТЕ-2 и для работы без операционной системы. Используется с 1985 г. Ядро занимает 14 К байт (300 слов). Система ориентирована на стандарт «Форт-83». В реализации использован прямой шитый код. Имеются своя файловая система, средства для связи с операционной системой, строковый редактор, ориентированный на файловую систему.

Форт-БЭСМ-6. Система разработана в Институте теоретической астрономии (ИТА) АН СССР И. Р. Агамирзяном для ЭВМ БЭСМ-6. Используется с 1984 г. Работает под управлением ОС ДИСПАК и имеет интерфейс с файловой системой КРАБ. Общий объем — 24 К байт (500 слов). По входному языку система ближе всего к стандарту «Фиг-форт», вместе с тем используется ряд слов из стандарта «Форт-83». Сравнительно большой объем памяти связан с отсутствием в ЭВМ БЭСМ-6 байтовой адресации. Для представления стандартного двухбайтного значения используется шестибайтное машинное слово БЭСМ-6. По той же причине для работы с байтовыми значениями вместо слов С@, С! и других введен ряд специальных слов. Из-за особенностей системы команд в реализации применен подпрограммный шитый код. Это позволяет включать ассемблерные вставки непосредственно в шитый код и наоборот — высокоуровневые слова как обращения к подпрограммам внутрь ассемблерных определений. Прототипом для встроенного ассемблера является ассемблер МАДЛЕН БЭСМ-6. В качестве текстового редактора используется стандартный редактор операционной системы.

Форт-Эльбрус. Система разработана на математико-механическом факультете ЛГУ А. Е. Соловьевым для МВК «Эльбрус». Работает с 1986 г. под управлением ОС «Эльбрус». Ядро занимает 4,5 К байт (200 слов). Система ориентирована на стандарт «Форт-83», имеет ряд специальных инструментальных слов и средства для связи с процедурами на языке Эль-76.

СПИСОК ЛИТЕРАТУРЫ

1. **Астановский А. Г.,** Ломунов **В. Н.** Процессор, ориентированный на язык Форт//Программирование микропроцессорной техники. Таллин: Ин-т кибернетики АН ЭССР, 1984. С. 50—67.

2. **Баранов С. Н.** Система Форт-ЕС. Л., 1986. 48 с. (Препринт ЛИИАН № 1).

3. **Баранов С. Н.** Стандарты языка Форт/ЛИИАН, 1987. 88 с. Деп. в ВИНИТИ. 04.06.87. ФН 4012-887.

4. **Баранов С. Н., Кириллин В. А.,** Ноздрунов Н. Р. Реализация языка Форт на дисплейном комплексе ЕС-7970//Программирование микропроцессорной техники. Таллин: Ин-т кибернетики АН ЭССР, 1984. С. 41—49.

5. **Берс А. А., Поляков В. Г., Руднев С. Б.** Основные принципы разработки системы программирования для микропроцессоров//Программное обеспечение задач информатики. Новосибирск: ВЦ СОАН, 1982. С. 5—25.

6. **Болдырев А. Ю., Веретенова Н. Н., Лезин Г. В.** ФОС: интерактивная система программирования для ЭВМ ИСКРА-226. Л., 1986. 74 с. (Препринт ИСЭП).

7. **Вулф А.** Операционные системы реального времени в русле развития вычислительной техники//Электроника. 1985. № 17. С. 46—56.

8. **Дейкстра** Э. Д. Взаимодействие последовательных процессов// Языки программирования/ Пер. с англ. М.: Мир, 1972. С. 9—86.

9. **Диалоговые** микропроцессорные системы/Под ред. Н. П. Брусенцова и А. М. Шаумана. М.: Изд-во МГУ, 1986. 148 с.

10. **Ершов А. П.** О сущности трансляции//Программирование. 1977. № 5. С. 21—39.

11. **Кнут Д.** Искусство программирования для ЭВМ/Пер. с англ. Т. I. М.: Мир, 1976. 735 с.

12. **Котляров В. П., Морозов Н. Б.** Технологические комплексы разработки программного обеспечения встроенных микроЭВМ// Индивидуальные диалоговые системы на базе микроЭВМ (персональные компьютеры) ДИАЛОГ-84-МИКРО. Л.: Наука, 1984. С. 93—96.

13. **Липаев В. В.** Тестирование программ. М.: Радио и связь, 1986. 296 с.

14. **Липаев В. В., Каганов Ф. А.,** Керданов А. В. Система автоматизации проектирования программ на базе персональных ЭВМ (система ПРА)//Микропроцессорные средства и системы. 1985. № 4. С. 42—45.

15. **Новиков Б. А.** Снова сопрограммы//Программирование. 1986. № 4. С. 59-62.

16. Новиков Б. А., Романовский И. В. Сопрограммы в ОС ЕС// Кибернетика. 1985. № 1. С. 34—37, 44.

17. **Руднев С. Б., Четвернин В. А.** Переносимая рабочая смесь — ядро программного обеспечения персональной ЭВМ//Персональные ЭВМ в задачах информатики. Новосибирск: ВЦ СОАН. 1984. С. 58—72.

18. **Сахаров** А. Л. Д/М— диалоговый язык для управления пакетами прикладных программ//Управляющие системы и машины. 1985. № 3. С. 105—109.

19. **Технология** проектирования компонентов программ АСУ/ В. В. Липаев, Л. А. Серебровский, П. Г. Гаганов и др. М: Радио и связь, 1983. 264 с.

20. Томбак М**. О., Пёял Я. Р., Соо В. К.** Использование расширяемого языка в СПТ//Тр. Вычислит. центра Тартус. ун-та. Вып. 52. 1985. с. 39—54.

21. Ballard B. FORTH Direct Execution Processors in the Hopkins Ultraviolet Telescope//J. FORTH App.. and Res. 1984. Vol. 2. N 1. P. 33—47.

22. **Dewar R. K.** Indirect Threaded Code//CACM. 1975. Vol. 18. N 6. P. 330—331.

23. **FORTH-83** Standard. Mountain View (USA): FORTH Standards Team, 1983. 82 p.

24. **Harris K.** The FORTH Philosophy//Dr. Dobb's J. 1981. Vol. 6. N 59. P. 6—11.

25. **Loeliger R. G.** Threaded Interpretative Languages. Peterbourgh: Byte Books, 1980. 251 p.

26. **McCabe C. K.** FORTH-83: Evolution Continues//BYTE. 1984. Vol. 9. N 8/ P. 137—138, 412—415, 418-421.

27. **Moore C H.** FORTH: A New Way to Program a Mini-Computer//Astr. and Astrophys. Suppl. 1974. Vol. 5. P. 497—511.

28. Ragsdale W. F. Fig-FORTH Installation Manual. Glossary. Model. Editor. San Carlos: FORTH Interest Group, 1980. 61 p.

29 **Ritter T., Walker G7** Varieties of Threaded Code for Language Implementation//BYTE. 1980. Vol. 5. N 9. P. 206—227.

30. **Tello** E. PolyFORTH and PC/FORTH//BYTE. 1984. Vol. 9. N 12. P. 303—310, 312, 314.

31. **Walker R. V., Rather E. D.** PolyFORTH II Reference Manual. S. L.: FORTH Inc., 1983. 524 p.

ОГЛАВЛЕНИЕ

Производственное издание

Сергей Николаевич **Баранов**
Николай Романович Ноздрунов

ЯЗЫК ФОРТ И ЕГО РЕАЛИЗАЦИИ

Редактор *Т. Г. Филатова*
Художественный редактор *Н. В. Зимаков*
Технический редактор *Т. М. Жилич*
Корректоры И. *Г. Иванова, Н. Б. Старостина*
Обложка художника *В. И. Коломейцева*

ИБ № 5745

Сдано в набор 11.05.87. Подписано в печать 10.12.87. М-18593. Формат 84x10 8/32. Бумага типографская № 2. Гарнитура литературная. Печать высокая. Усл. печ. л. 8,4. Усл. кр.-отт. 8,61. Уч-изд. л. 7,84. Тираж 100 000 экз. (2-й завод 50 001 — 100 000). Заказ 966. Цена 1 р. 20 к.

Ленинградское отделение ордена Трудового Красного Знамени издательства «Машиностроение». 191065, Ленинград, ул. Дзержинского, 10.

Ленинградская типография № 2 головное предприятие ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств , полиграфии и книжной торговли 198052, Ленинград, Л-52. Измайловский проспект, 29.

Уважаемый читатель!

В целях получения информации о качестве наших изданий просим Вас в прилагаемой анкете подчеркнуть позиции, соответствующие Вашей еценке этой книги.

1. В книге существует:
   а) острая необходимость
   б) значительная потребность
   в) незначительная потребность

2. Эффективность книги с точки зрения практического вклада в отрасль:
   а) весьма высокая
   б) высокая
   в) сомнительная
   г) незначительная

ЛИНИЯ ОТРЕЗА

3. Эффективность книги с точки зрения теоретического вклада в отрасль:
   а) весьма высокая
   б) высокая
   в) сомнительная
   г) незначительная

4. Материал книги соответствует достижениям мировой науки и техники в данной отрасли:
   а) в полной мере
   б) частично
   в) слабо

5. Книга сохранит свою актуальность:
   а) 1—2 года
   б) в течение 5 лет
   в) длительное время

6. Название книги отвечает содержанию:
   а) в полной мере
   б) частично
   в) слабо

Дополнительные замечания предлагаем Вам приложить отдельно.

Фамилия, имя, отчество

Ученое звание . . . .

Специальность . . . .

Место работы, должность

Стаж работы . . . .

Просим отрезать страницу по линии отреза и в почтовом конверте выслать по адресу: 191065, Ленинград, ул. Дзержинского, 10, ЛО изд-ва «Машиностроение»

С. Н. Баранов, Н. Р. Ноздрунов. Язык Форт и его реализации

ЛИНИЯ ОТРЕЗА



ЯЗЫК ФОРТ И ЕГО РЕАЛИЗАЦИИ

1 р. 20 к.

Язык Форт появился в начале 1970-х гг. в США. Быстрый рост его популярности приходится на середину 1970-х гг., когда появились персональные ЭВМ. Широкое распространение получили коммерческие программные продукты, написанные на языке Форт — системы обработки текстов, пакеты машинной графики, трансляторы, видеоигры. В 1983 г. опубликован стандарт "Форт-83", в соответствии с которым строится изложение в данной книге.

Система программирования на Форте (форт-система) обычно обеспечивает полный набор средств поддержки для разработки и исполнения программ: операционную систему, интерпретатор для диалогового исполнения, компилятор, ассемблер, текстовый редактор и обслуживающие программы. Все эти компоненты являются расширениями Форта и написаны на том же Форте.

Таким образом, Форт — это и система для пользователя и метасистема. В соответствии с этим принципом в форт-системе используется минимум правил и накладывается минимум ограничений, т. е. почти нет синтаксиса, жестко контролируемых интерфейсов для взаимодей:твия модулей, закрытых для пользователя областей памяти, имеется лишь незначительный по объему встроенный контроль ошибок.

